NL4Py: Agent-Based Modeling in **Python** with Parallelized **NetLogo** Workspaces

Chathika Gunaratne
Complex Adaptive Systems Lab
University of Central Florida

Ivan Garibay
Complex Adaptive Systems Lab
University of Central Florida

Abstract

NL4Py is a NetLogo controller software for Python, for the rapid, parallel execution of NetLogo models. **NL4Py** provides both headless (no graphical user interface (GUI)) and GUI NetLogo workspace control through Python. Spurred on by the increasing availability of open-source computation and machine learning libraries on the Python package index, there is an increasing demand for such rapid, parallel execution of agent-based models through Python. NetLogo, being the language of choice for a majority of agent-based modeling driven research projects, requires an integration to Python for researchers looking to perform statistical analyses of agent-based model output using these libraries. Unfortunately, until the recent introduction of **PyNetLogo**, and now **NL4Py**, such a controller was unavailable.

This article provides a detailed introduction into the usage of **NL4Py** and explains its client-server software architecture, highlighting architectural differences to **PyNetLogo**. A step-by-step demonstration of global sensitivity analysis and parameter calibration of the Wolf Sheep Predation model is then performed through **NL4Py**. Finally, **NL4Py**'s performance is benchmarked against **PyNetLogo** and its combination with **IPyParallel**, and shown to provide significant savings in execution time over both configurations.

Keywords: **NL4Py**, Python, NetLogo, agent-based modeling, individual-based modeling, complex adaptive systems, global sensitivity analysis, parameter calibration, **SALib**, **DEAP**.

1. Introduction

Agent-based modeling (ABM), also referred to as individual-based modeling (IBM), is a modeling and simulation technique where the outcome of a system, or macro-behavior, is modeled as the result of the interactions of independently acting, decentralized micro-level agents/individuals. A popular technique in complex adaptive systems research (Mitchell 2009; Farmer and Foley 2009; Janssen and Ostrom 2006; Axtell 2008), agent-based models are able to reproduce emergent phenomena, dynamic equilibria, and non-linear outcomes, properties that are observed in the actual complex systems that they represent. Furthermore, ABM outputs are highly sensitive to initial conditions, often producing a vast space of possible outcomes (Lee *et al.* 2015) that require rigorous statistical scrutiny to untangle. Two such methodologies often accompanying ABMs are sensitivity analysis and optimization, more specifically known as parameter calibration.

1.1. NetLogo

At the time of writing, NetLogo (Wilensky and Others 1999; Wilensky and Rand 2015) is the most popular ABM toolkit used in the socio-ecological modeling community (see Fig. 1). As ABM has seen a rapid rise in application across several domains, the usage of NetLogo has also increased. NetLogo is a simple, dynamically-typed, pseudo-functional programming language, influenced by Logo and Lisp, accompanied with a drag-and-drop user interface builder for parameter controls and model output display, making it quick and easy to learn. For this reason, NetLogo has played a crucial role in the computational social sciences in particular, enabling researchers with modest, or no, programming skills to quickly develop and experiment with agent-based models. While many other agent-based modeling frameworks exist with varying popularity (eg. Repast Simphony/HPC (North *et al.* 2013; Collier and North 2013), MASON (Luke *et al.* 2005), AnyLogic (Borshchev 2013)), NetLogo has remained the platform of choice for many conducting agent-based modeling research.
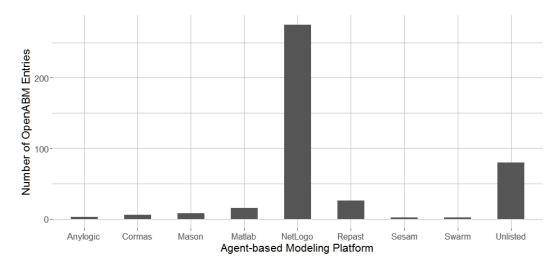

Figure 1: Platform by Agent-based models on OpenAbm, now CoMSES.net (CoMSES 2018 (Accessed May 29,2018), between 2008 and 2018. NetLogo is clearly dominant.

1.2. Link to statistics

Similar to the complex systems they represent, ABM simulations are highly sensitive to their parameter configurations. Often, the target macro-behavior may or may not be observed depending on the provided parameter values. In other words, an agent-based model accurately capturing the essence of a real-world system can still fail to achieve a desired output if the the correct parameter values are not provided or are unknown. Knowing the parameter values, or estimated range of values, can be vital to policy design, especially if the intention is to target a low-frequency, high impact macro-behavior, i.e., an anomaly. For example, under what conditions does a stock market crash occur in a model of independently acting trading agents?

Due to this reason, the agent-based modeling community has shown increasing interest in exploring this parameter space. These explorations range from simple factorial design experiments, to single or multi-objective optimization (Stonedahl and Wilensky 2010), to sensitivity

analysis (Lee *et al.* 2015; Ligmann-Zielinska *et al.* 2014), model/rule discovery (Gunaratne and Garibay 2017), and identification of unique patterns (Chérel *et al.* 2015). For these reasons it has become increasing desirable to make NetLogo 'controllable' through scripting languages that are rich with quick and easy-to-use statistical, computational, and machine learning packages.

1.3. Python

Python is such a candidate language, with an extensive, well-maintained collection of machine learning and computation libraries. A few commonly used examples are: **Numpy** (Oliphant 2006), **SciPy** (McKinney 2010), **Pandas** (McKinney 2011), **SALib** (Herman and Usher 2017), **Scikit-learn** (Pedregosa *et al.* 2011), **DEAP** (Fortin *et al.* 2012), and **Matplotlib** (Hunter 2007) to name a few. Additionally, Python's ease of use and short learning curve paired with browser-based integrated development environments (IDEs) such as **Jupyter** notebook (Kluyver *et al.* 2016) has made it the language of choice for many looking to perform quick, data intensive experiments.

Taking into account the above reasons, there is an increasing need for the agent-based modeling community to have access to a NetLogo controlling package from Python for model output analysis. NetLogo itself is written in a combination of Java and Scala, making it difficult to directly call and control NetLogo procedures through Python. Instead, Python developers have to access the NetLogo controlling application programming interface (API) provided with NetLogo. This process is not straightforward and requires access to the Java native interface (JNI) or remote procedure calls to the Java virtual machine (JVM) through sockets. Importantly, many statistical techniques used to analyze ABM output require vast counts of model runs under varying model configurations.

A package able to handle these requirements, by providing an API for quick and easy access to NetLogo through Python, and able to handle the concurrency issues and optimization of computational resource use for executing many parallel model runs, is required. Such a library exists for R, **RNetLogo** (Thiele 2014). A Python solution for this problem was recently introduced as **PyNetLogo** (Jaxa-Rozen *et al.* 2018), providing developers in Python the ability to start and control NetLogo models in both Headless (no graphical user interface (GUI) mode for fast execution) and GUI-enabled modes. However, the workflow necessary to execute parallel instances of NetLogo headless workspaces via **PyNetLogo**, as demonstrated by the authors, requires IPython (Pérez and Granger 2007) and setting up an **IPyParallel** (IPython Development Team 2018) cluster, external to the Python code. The headless workspaces are then submitted as jobs into the **IPyParallel** engines to be executed in parallel. This method requires decorations such as %%p, that necessitate an IPython environment and **Jupyter** Notebook making the provided examples difficult to run as native Python script. Considering that many researchers using ABMs as a methodology are from domains other than Computer Science, a NetLogo controller for Python with a simpler workflow and able to internally run multiple workspaces in parallel, is desired.

In this paper, we introduce NL4Py **NL4Py**, a NetLogo controller for Python, developed with the goals of usability, rapid parallel execution, and model parameter access in mind. In addition, **NL4Py** is platform independent, supporting Windows, MacOS, and Linux, and supports both Python 2 and 3. Unlike **PyNetLogo**, which uses the Java native interface (JNI) framework to access the JVM, **NL4Py**, inspired by (Masad 2016), employs a client-server architecture via

Py4J (Dagenais 2009–2015), a Python-Java bridging package. **NL4Py** automatically downloads and hosts a `NetLogoControllerServer` Java archive (JAR) executable that handles the parallel execution of NetLogo workspaces internally as Java threads. The client-server architecture allows **NL4Py** to package and hide the excessive programming required to maintain multiple parallely executing NetLogo models, which may even have to be queried regularly depending on the usecase. Further, the `NetLogoControllerServer` ensures thread safety and handles JVM memory allocation/garbage collection, reducing these burdens from the Python application developer. In short, **NL4Py** performs the parallelization of NetLogo workspaces on the JVM, instead of leaving it to the user's Python application, unlike **PyNetLogo**.

This paper is written as both a scientific article and a technical document explaining **NL4Py** and is organized as follows. In section 2, **NL4Py** installation, requirements, and functions are described. Section 3 demonstrates two use cases of **NL4Py**, sensitivity analysis and calibration on the Wolf Sheep Predation model (**?**), and discusses the model's sensitivity to parameters and finding of a short-term near-equilibrium state via calibration. Section 4 explains **NL4Py**'s software architecture. In section 5, we present the execution time improvements made by **NL4Py** when running parallel runs of three NetLogo sample models over the execution times of **PyNetLogo** and **PyNetLogo** with **IPyParallel**. Finally, in sections 6 and 7 we discuss known limitations of **NL4Py** and conclude.

2. Controlling **NetLogo** in **Python** with **NL4Py**

In this section, we describe how users can setup **NL4Py** and control multiple NetLogo model runs from within their Python script with the help of **NL4Py**.

2.1. Installation

NL4Py is made available on the Python package index (Python Software Foundation Accessed 29 May, 2018) for easy installation and version control. At the time of writing, **NL4Py** is in release version 0.5.0 (Gunaratne 2018). **Pip** tools (Python's package manager) can be used to install **NL4Py** using the following command:

```
>>> pip install NL4Py
```

2.2. Requirements

NL4Py works with NetLogo 6.0.2 or higher and requires Java development kit (JDK) 8 or higher to be installed. Other Python dependencies such as **Py4J** will be installed automatically with **pip** tools. **NL4Py** has been tested on both Python 2.7 and Python 3.6 on Windows 10, MacOSX 10.10 and Ubuntu operating systems.

2.3. Using the NL4Py API

NL4Py allows both NetLogo `HeadlessWorkspace` creation and control, and also NetLogo GUI-enabled application control. In this section, we describe how users can control NetLogo in both GUI and headless modes with **NL4Py**.

Starting and stopping the `NetLogoControllerServer`

The first step to controlling NetLogo through **NL4Py** is by importing **NL4Py** and starting the `NetLogoController` server. This can be done with the following commands:

```
>>> import nl4py
>>> nl4py.startServer(path_to_netlogo)
```

The function requires the path to the top level directory of the NetLogo installation as a string argument. The `NetLogoControllerServer` is then started and ready for requests from the **NL4Py** client through Python for NetLogo controlling. In complement to this, the `NetLogoControllerServer` can be shutdown, in order to free computational resources using the following command:

```
>>> nl4py.stopServer()
```

Using NetLogo in GUI mode

In order to start and control the NetLogo application in GUI mode, users can execute the following command in their Python script:

```
nl4py.NetLogoApp()
```

Users can then use the `NetLogoApp()` functions to send commands, execute reporters, and schedule reporters to the NetLogo Application. These functions are listed in Tab. 1.

Using NetLogo headless workspaces

However, most optimization work requires vast parallel runs of NetLogo models and GUI mode unnecessarily burdens computational resources during such an analysis. NetLogo provides headless workspaces for these purposes, which are essentially NetLogo models running without the GUI enabled. `HeadlessWorkspaces` tend to run more efficiently, since there is no computation required for rendering visualizations **BehaviorSearch**, the model calibration tool that is packaged with NetLogo, uses headless workspaces driven by optimization algorithms and is a prime example of their utility.

NL4Py provides API controls for Python developers to create NetLogo headless workspaces, open and close models on these workspaces, get and set parameters to the models, send NetLogo commands these models, and schedule and execute reporters to query the simulation state at regular intervals or at the end of a simulation run. Resulting `NetLogoHeadlessWorkspace` objects can then be used to open and control NetLogo models from within Python.

`NetLogoHeadlessWorkspace`s can be created with the following function:

```
nl4py.newNetLogoHeadlessWorkspace()
```

Additionally, users can get a list of all the existing `NetLogoHeadlessWorkspace`s, delete all the existing `NetLogoHeadlessWorkspace`s, and delete a single `NetLogoHeadlessWorkspace` with the following commands, respectively:

```
nl4py.deleteAllHeadlessWorkspaces()
nl4py.getAllHeadlessWorkspaces()
nl4py.deleteHeadlessWorkspace(nl4py.NetLogoHeadlessWorkspace)
```

Opening and closing models

The following commands can be then used to open and close models on
`NetLogoHeadlessWorkspace`s, respectively:

```
nl4py.NetLogoHeadlessWorkspace.openModel("path_to_model")
nl4py.NetLogoHeadlessWorkspace.closeModel()
```

Similarly, the same can be done on the NetLogo GUI application with the following:

```
nl4py.NetLogoGUI.openModel(path_to_model)
nl4py.NetLogoGUI.closeModel()
```

Commands and basic reporters

NL4Py provides users the ability to execute NetLogo commands and reporters from within
their Python application. The `command` function takes NetLogo syntax as strings and ex-
ecutes the command on the respective workspace. The `report` function takes in NetLogo
syntax as strings, executes the reporter, and returns the results. In the case of a failed re-
porter due to a NetLogo exception, `report` will report the exception. Return values from
`report` are typically strings and must be cast into their correct data types accordingly. On
`NetLogoHeadlessWorkspace`s the following execute commands and reporters:

```
nl4py.NetLogoHeadlessWorkspace.command(netlogo_command_string)
nl4py.NetLogoHeadlessWorkspace.report(netlogo_reporter_string)
```

Similarly, the equivalent for NetLogo GUI application control:

```
nl4py.NetLogoGUI.openModel(path_to_model)
nl4py.NetLogoGUI.closeModel()
```

Working with parameters

NL4Py allows users to query a NetLogo model's parameter names, get the suggested ranges
as set on the NetLogo interface objects (slider min/max values, list values, etc), and set
the parameters to random values. The three following methods provide these functions for
NetLogoHeadlessWorkspaces, respectively:

```
nl4py.NetLogoHeadlessWorkspace.setParamsRandom()
nl4py.NetLogoHeadlessWorkspace.getParamNames()
nl4py.NetLogoHeadlessWorkspace.getParamRanges()
```

Similarly, these functions are available for the NetLogo GUI application:

```
nl4py.NetLogoGUI.setParamsRandom()
nl4py.NetLogoGUI.getParamNames()
nl4py.NetLogoGUI.getParamRanges()
```

Reporter scheduling

In certain instances, a user may require to record the simulation state at regular intervals, often for every simulation tick, over a given period of time. For this, **NL4Py** provides scheduled reporters. Multiple reporters can be specified as a Python list of strings of NetLogo commands. This reporter list along with the start tick, stop tick, and interval of ticks required between each reporter execution can be passed into into `scheduleReporterAndRun` for this purpose. Optionally, a custom `go` command can be supplied to `scheduleReportersAndRun` in the case that the NetLogo model's execution procedure has a name different to the standard go. **NL4Py** then schedules the reporters to execute on the respective NetLogo workspaces and store results on the `NetLogoControllerServer` from start time till stop time at every interval number of ticks.

On `NetLogoHeadlessWorkspace`s, this method signature is:

```
nl4py.NetLogoHeadlessWorkspace.scheduleReportersAndRun(reporters_array, \
    startAtTick = 0, intervalTicks = 1, stopAtTick = -1, goCommand = "go")
```

Similarly, on the NetLogo GUI Application mode this is:

```
nl4py.NetLogoGUI.scheduleReportersAndRun(reporters_array, startAtTick = 0, \
    intervalTicks = 1, stopAtTick = -1, goCommand = "go")
```

The results stored on the `NetLogoControllerServer` can be queried at anytime during or after the scheduled reporters execution, with `getScheduledReporterResults`. This function returns a **Numpy** array of lists of the reporter results, for each execution thus far. This function is non-blocking to prevent imposing unnecessary wait times on the user's application. If the model has not finished execution, then an empty array will be returned. On `NetLogoHeadlessWorkspace`s this function is:

```
nl4py.NetLogoHeadlessWorkspace.getScheduledReporterResults()
```

Similarly, in the NetLogo GUI application:

```
nl4py.NetLogoGUI.getScheduledReporterResults()
```

3. Applications

In this section, we demonstrate how libraries available in the Python package index (Python Software Foundation Accessed 29 May, 2018) can be used in combination with **NL4Py** to conduct statistical experimentation and optimization of agent-based models in NetLogo. In

Function	Parameters	Return Value	Description
nl4py			
`startServer(path_to_netlogo)`	-	-	Starts the `NetLogoControllerServer`
`stopServer()`	-	-	Stops the `NetLogoControlerServer`
`newNetLogoHeadlessWorkspace()`	-	`NetLogo-HeadlessWorkspace`	`NetLogoControllerServer` creates a new `HeadlessWorkspace` and returns its controller for access from **Python**.
`deleteAllHeadlessWorkspaces()`	-	-	Deletes all the existing workspaces and their controllers currently on the `NetLogoControllerServer`.
`getAllHeadlessWorkspaces()`	-	List of all `HeadlessWorkspace` objects	Returns a **Python** list of all the `NetLogoHeadlessWorkspace` objects known by the client.
`deleteHeadlessWorkspace( nl4py. NetLogo-HeadlessWorkspace )`	`NetLogo-HeadlessWorkspace`	-	Deletes a `HeadlessWorkspace()`.
`NetLogoApp()`	-	`NetLogoGUI` object	Brings up the **NetLogo** Application in GUI mode
nl4py.NetLogoHeadlessWorkspace and nl4py.NetLogoGUI			
`openModel(modelPath)`	**NetLogo** file path	-	Opens the **NetLogo** model from the specified file within the `NetLogoHeadlessWorkspace` or **NetLogo** Application.
`closeModel()`	-	-	Closes the opened model.
`command(command)`	**NetLogo** command string	-	Executes command on opened model. Function is non-blocking: returns immediately.
`report(reporter)`	**NetLogo** reporter string	String	Executes a reporter on the opened model and returns the result as a string. Function is non-blocking: returns immediately.
`setParamsRandom()`	-	-	Sets all the parameters on the **NetLogo** interface to random values.
`getParamNames()`	-	List of strings	Returns the names of the parameters of the model from the **NetLogo** interface.
`getParamRanges()`	-	List of floats	Returns the parameter ranges specified by the interface components of the model.

Table 1: Summary of functions offered by the **NL4Py** API.

particular, we illustrate two commonly used techniques in the agent-based modeling community, sensitivity analysis and calibration. **SALib**'s Sobol sampling and analysis was conducted for sensitivity analysis (Herman and Usher 2017). The **DEAP** package was used to run an evolutionary algorithm for the calibration experiment (Fortin *et al.* 2012).

We performed sensitivity analysis and calibration both on the Wolf Sheep Predation model, a model of population dynamics of a producer, predator, and prey ecosystem (**?**) as an ABM alternative to the Lokta-Volterra equations (Lotka 1926; Volterra 1926). The model has 5 parameters related to the sheep, wolves, and grass version, namely sheep-gain-from-food, sheep-reproduce, wolf-gain-from-food, wolf-reproduce, and grass-regrowth-time, and two initial conditions initial-number-sheep, and initial-number-wolves. The model studies the stability of the populations over time. An equilibrium state, can occur under two conditions: 1) the populations are extinct, or 2) there is minimal change over time for both populations of sheep and wolves.

For both sensitivity analysis and calibration we quantified and scored closeness to equilibrium as the total population stability, E_t, at every simulation time step, t. E_t was found as follows. We measured the first order derivatives of both the wolf population (P_W) and sheep population (P_S) over time, $\frac{dP_{W,t}}{dT}$ and $\frac{dP_{S,t}}{dT}$, by finding the change in population between each simulation tick (see Eqs. 1 and 2). These derivatives were then passed through a unit step function, in order to score extinction of either species as 0. The reciprocal of this result was considered the population stability score for each species at t, $E_{W,t}$ and $E_{S,t}$ (see Eq. 3). We then took the mean population stability for both species, for each simulation step, to get a total population stability over time. Finally, an aggregate mean total population stability score, *score*, was found by dividing the result by the number of time steps simulated, k (see Eq. 4).

$$\frac{dP_{W,t}}{dT} = ((P_{W,t} - P_{W,t-1})/\Delta T \quad and \quad \frac{dP_{S,t}}{dT} = ((P_{S,t} - P_{S,t-1})/\Delta T \tag{1}$$

$\Delta T = 1$, since the ABM runs in discrete simulation time units, or ticks. So,

$$\frac{dP_{W,t}}{dT} = ((P_{W,t} - P_{W,t-1}) \quad and \quad \frac{dP_{S,t}}{dT} = ((P_{S,t} - P_{S,t-1}) \tag{2}$$

$$E_{W,t} = \begin{cases} \frac{1}{dP_{W,t}}, & \text{if } P_{W,t} \geq 1 \\ 0, & \text{otherwise} \end{cases} \quad and \quad E_{S,t} = \begin{cases} \frac{1}{dP_{S,t}}, & \text{if } P_{S,t} \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$score = \frac{\sum_{t=0}^{k}(E_{S,t} + E_{W,t})}{k} \tag{4}$$

3.1. Calibration with NL4Py

Calibration is a commonly used technique in agent-based modeling for finding the parameter configuration(s) required to produce a particular target output. Typically, any optimization algorithm can be used to calibrate an agent-based model. Calibration has become such an integral part of the methodology that agent-based modeling platforms have been accompanied with ready-made calibration tools, such as **OptQuest** for AnyLogic and **BehaviorSearch** for NetLogo. However, if the user desires to use an optimization algorithm outside of the provided calibration tool, they will need a means from which to control the agent-based model from their own custom optimization software.

The Python package index has several optimization packages, from which we choose **DEAP** (Fortin *et al.* 2012), mainly due to its popularity and ease of use. **DEAP** is also easily parallelizable, a feature used in the example in this section. We used **DEAP** to configure and run an evolutionary algorithm (EA) provided by the package to calibrate the Wolf Sheep Predation model.

The objective of the calibration process in this experiment was finding a parameter configuration that was able to closely simulate equilibrium, without extinction, in the Wolf Sheep Predation model, described earlier in this section (see Eqs. 1 , 2, 3, 4).

First, parameter names and ranges of the model were read in from the NetLogo file using **NL4Py** and used to define an Individual in the EA:

```
import random
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
creator.create("FitnessMax", base.Fitness, weights = (1.0,))
creator.create("Individual", list, fitness = creator.FitnessMax)
toolbox = base.Toolbox()
import nl4py
nl4py.startServer("/home/ubuntu/NetLogo 6.0.3/")
n = nl4py.newNetLogoHeadlessWorkspace()
n.openModel("Wolf Sheep Predation.nlogo")
parameterNames = n.getParamNames()
parameterRanges = n.getParamRanges()
parameterInitializers = []
for parameterName, parameterRange in zip(parameterNames, parameterRanges):
    parameterName = ''.join(filter(str.isalnum, str(parameterName)))
    if len(parameterRange) == 3:
        toolbox.register(parameterName, random.randrange, \
            parameterRange[0], parameterRange[2], parameterRange[1])
        parameterInitializers.append(eval("toolbox." + str(parameterName)))
toolbox.register("individual", tools.initCycle, creator.Individual,
tuple(parameterInitializers))
```

This EA Individual description was then used to define the population for the EA using the **DEAP** toolbox:

```
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("mate", tools.cxTwoPoint)
lowerBounds = [row[1] for row in parameterRanges[: -2]]
upperBounds = [row[2] for row in parameterRanges[: -2]]
toolbox.register("mutate", tools.mutUniformInt, low = lowerBounds, \
        up = upperBounds, indpb = 0.1)
toolbox.register("select", tools.selTournament, tournsize = 3)
```

Next, a `simulate` function was defined to use **NL4Py** to initialize, run, and measure the aggregate mean total population stability according to Eqs. 1, 2, 3, and 4, as follows:

```
import time
import pandas as pd
import numpy as np
def simulate(workspace_, names,values):
    workspace_.command("stop")
    for name, value in zip(names, values):
        cmd = 'set {0} {1}'.format(name, value)
        workspace_.command(cmd)
    workspace_.command('set model-version "sheep-wolves-grass"')
    workspace_.command('setup')
    workspace_.scheduleReportersAndRun( \
            ["ticks", 'count sheep' , 'count wolves'], 0, 1, 500, "go")
    newResults = []
    while(len(newResults) == 0):
        newResults = workspace_.getScheduledReporterResults()
        if len(newResults) > 0:
            ###Process simulation results###
            df = pd.DataFrame(newResults)
            sheep_pop = pd.to_numeric(df.iloc[:, 1])
            wolves_pop = pd.to_numeric(df.iloc[:, 2])
            dsheep_dt = sheep_pop.diff().abs()
            dwolves_dt = wolves_pop.diff().abs()
            population_stability_sheep = np.divide(1, (dsheep_dt + \
                0.000001)).mul(np.where(sheep_pop == 0, 0, 1))
            population_stability_wolves = np.divide(1, (dwolves_dt + \
                0.000001)).mul(np.where(wolves_pop == 0, 0, 1))
            population_stability_total = (population_stability_sheep \
                + population_stability_wolves) / 2
            aggregate_metric = population_stability_total.sum() \
                                / len(population_stability_total)
            ###Done processing simulation results###
            workspace_.command("stop")
            return aggregate_metric,
        time.sleep(2)
```

Note that 0.000001 was added to the denominator when calculating the population stability of each species in addition to Eq. 3. This was to avoid a division by zero error. This also gave us a practical upper-bound to *score* at 10^7. Next, **NL4Py** was used to create `NetLogoHeadlessWorkspace`s corresponding to each EA `individual`:

```
nl4py.deleteAllHeadlessWorkspaces()
POP = 200
freeWorkspaces = []
for i in range(0, POP):
```

```
n = nl4py.newNetLogoHeadlessWorkspace()
n.openModel('Wolf Sheep Predation.nlogo')
freeWorkspaces.append(n)
```

Next, the evaluation function of each EA `individual` was set as the `simulate` function above:

```
def evaluateWolfSheepPredation(individual):
    n = freeWorkspaces[0]
    freeWorkspaces.remove(n)
    result = simulate(n,parameterNames,individual)
    freeWorkspaces.append(n)
    return result
toolbox.register("evaluate", evaluateWolfSheepPredation)
```

Due to the massive number of model runs required (100 generations of around 200 model evaluations), the calibration experiment was run on all cores of an Intel(R) Core i7-7700 CPU (7th generation) with 16GB of RAM running Windows 10 (64 Bit). **DEAP** was configured to run on all available cores by registering the `map` function into the **DEAP** `toolbox` to utilize the **ThreadPool** provided by Python's **Multiprocessing** library, as follows:

```
import multiprocessing
from multiprocessing.pool import ThreadPool
pool = ThreadPool(multiprocessing.cpu_count())
toolbox.register("map", pool.map)
```

The EA was configured to use a two-point crossover, a crossover rate of 0.8, a mutation rate of 0.2, population size of 200 individuals, store the best individual selected for the hall of fame, and run for 100 generations:

```
stats = tools.Statistics(key = lambda ind: ind.fitness.values)
stats.register("max", np.max)
stats.register("mean", np.mean)
hof = tools.HallOfFame(1)
final_pop, log = algorithms.eaSimple(toolbox.population(n = POP), toolbox, \
    cxpb = 0.8, mutpb = 0.2, ngen = 100,stats = stats,halloffame = hof)
```

The EA was able to quickly converge towards an optimal solution with a high fitness (900000). The progress of the calibration was plotted using **Matplotlib** as shown in Fig. 2, using the following code:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['font.size'] = 25.0
plot = convergence_progress.plot(legend = True)
plt.xlabel("Generation")
plt.ylabel("Fitness")
fig = plot.get_figure()
fig.set_size_inches(18.5, 10.5)
fig.savefig("CalibrationConvergenceProgress.png")
```
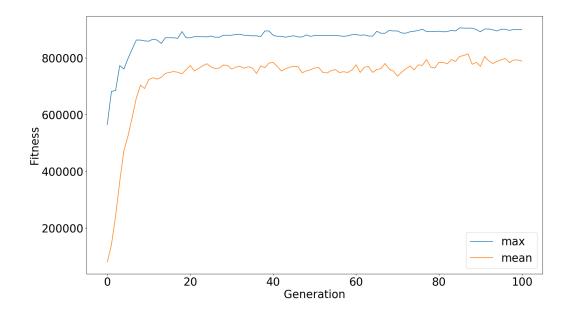
Figure 2: Max fitness (blue) of the best individual and mean fitness(orange) over generation of the evolutionary algorithm calibrating the Wolf Sheep Predation model towards equilibrium.

The parameter configuration of the best individual was as follows: initial-number-sheep = 236, sheep-gain-from-food = 3, sheep-reproduce = 1, initial-number-wolves = 47, wolf-gain-from-food = 92, wolf-reproduce = 0, and grass-regrowth-time = 97.

We then visualized the simulation output as in Fig. 3 using the **NL4Py** `NetLogoApp` function:

```
app = nl4py.NetLogoApp()
app.openModel("./Wolf Sheep Predation.nlogo")
parameterNames = n.getParamNames()
#hof, the hall of fame has the calibrated parameters of the best individual
for name, value in zip(parameterNames, hof[0]):
    app.command('set {0} {1}'.format(name, value))
app.command("setup")
app.command("repeat 1000 [go]")
```

It was seen that when running the best parameters in GUI mode, the wolf population remained in equilibrium, while the sheep population dropped steadily. For this parameter configuration, the wolves were highly efficient at gaining nutrition from consuming sheep, yet could not reproduce, allowing for a constant population over time. However, this meant that the sheep population decreased slowly, as they were gradually hunted by the wolves. Despite, remaining in a near-equilibrium state up to the time period the model was calibrated for, simulations quickly fell out of equilibrium soon after this period had passed. This demonstrates the difficulty of achieving an exact equilibrium state of both species over a long period of time.
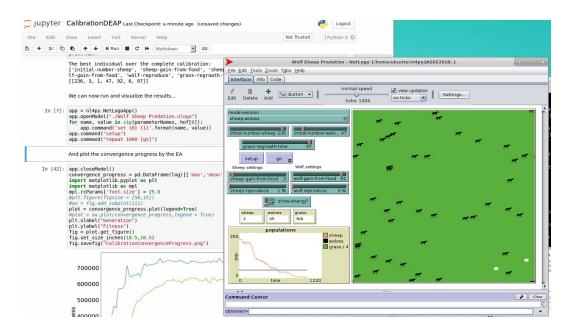
Figure 3: The Wolf Sheep Predation model run in GUI mode via **NL4Py** for the best parameters obtained through calibration to achieve a near-equilibrium state over 500 ticks. The wolf population is seen to remain constant over time, while the sheep number dropped gradually.

3.2. Sensitivity analysis with NL4Py

Agent-based models are highly sensitive to initial conditions and parameters similar to the complex systems they model. Accordingly, it is important to investigate the degree to which the model parameters affect the simulation output, and for what ranges of values the model produces reasonable results. This investigation is done through global sensitivity analysis.

As demonstrated in the previous section, achieving long term equilibrium between the two species in the Wolf Sheep Predation model is quite difficult. This phenomenon indicates that the equilibrium between the two species is highly sensitivity to one or more of the parameters of the model, and/or their high-order interactions. we performed a global sensitivity analysis in order to identify which parameters that the equilibrium state was most sensitive to.

Typically, sensitivity analysis involves sampling the n-dimensional parameter space, executing the agent-based model under these parameters, and analyzing the space of results produced by the model under these conditions.

As a result, a successful sensitivity analysis depends on the the quality of the parameter sampling technique. The most straightforward sampling technique, Monte-carlo sampling, performs a uniform random sampling. However, randomly sampling the parameter space can lead to biases. Low-discrepency methods, such as Sobol sensitivity analysis with Saltelli sampling (Sobol 2001; Saltelli 2002; Saltelli *et al.* 2010), or FAST (Cukier *et al.* 1973; Saltelli *et al.* 1999) are able to construct a more regular sample of parameter values.

NL4Py was used in combination with **SALib** to demonstrate the sensitivity of the variance of the equilibrium state of the model to the model parameters using Sobol sensitivity analysis with Saltelli sampling under different parameter sample sizes. Then we compared the sensitivity analyses results under the different sample sizes. Finally, we show how Python libraries, **Matplotlib** in particular, can be used to easily plot the sensitivity analysis results for visual

comparison. This analysis took 7000, 14000, and 21000 model evaluations for sample sizes of 500, 1000, and 1500, respectively, and was parallized by**NL4Py**, as demonstrated below.

The first step was to import **NL4Py** and start the `NetLogoControllerServer`:

```
import nl4py
nl4py.startServer("C:/Program Files/NetLogo 6.0.3")
```

We then define a `simulate` function that takes a `NetLogoHeadlessWorkspace` and a parameter setting, initializes, schedules reporters, and runs the Wolf Sheep Predation model for 100 ticks. The reporters scheduled record the simulation tick, the current number of sheep, and the current number of wolves:

```
def simulate(workspace_,parameters):
    workspace_.command("stop")
    for i, name in enumerate(problem['names']):
        if name == 'random-seed':
            workspace_.command('random-seed {}'.format(parameters[i]))
        else:
            workspace_.command('set {0} {1}'.format(name, parameters[i]))
    workspace_.command('set model-version "sheep-wolves-grass"')
    workspace_.command('set initial-number-sheep 100')
    workspace_.command('set initial-number-wolves 100')
    workspace_.command('setup')
    workspace_.scheduleReportersAndRun( \
        ["ticks",'count sheep','count wolves'], 0,1,100,"go")
```

Next, we define the `runForParameters` function that takes in the entire parameter space sample, from the sampling technique used, and executes `simulate` on each configuration, making sure that an optimal number of `NetLogoHeadlessWorkspace`s (i.e., same as the number of cores on the machine) continue to execute in parallel. **SALib** produces a total of 43000 model evaluations in total for the three sample sizes to test. To handle this number we use Python's **Multiprocessing** library to utilize all processor cores available on the machine. This function checks the workspaces and if the models runs are finished, calculates and returns the aggregate metric of total population stability, *score*, according to Eqs. 1, 2, 3, and 4 as the model output:

```
import pandas as pd
import numpy as np
def runForParameters(experiment):
    runsDone = 0
    runsStarted = 0
    runsNeeded = experiment.shape[0]
    aggregate_metrics = []
    import multiprocessing
    parallel_workspace_count = multiprocessing.cpu_count()
    nl4py.deleteAllHeadlessWorkspaces()
    for i in range(0, parallel_workspace_count):
```

```
        workspace = nl4py.newNetLogoHeadlessWorkspace()
        workspace.openModel('Wolf Sheep Predation.nlogo')
        simulate(workspace, experiment[runsStarted])
        runsStarted = runsStarted + 1
    while (runsDone < runsNeeded):
        for workspace in nl4py.getAllHeadlessWorkspaces():
            newResults = workspace.getScheduledReporterResults()
            if len(newResults) > 0:
                ###Process simulation results###
                df = pd.DataFrame(newResults)
                sheep_pop = pd.to_numeric(df.iloc[: , 1])
                wolves_pop = pd.to_numeric(df.iloc[: , 2])
                dsheep_dt = sheep_pop.diff().abs()
                dwolves_dt = wolves_pop.diff().abs()
                population_stability_sheep = np.divide( 1, (dsheep_dt \
                    + 0.000001)).mul(np.where(sheep_pop == 0, 0, 1) )
                population_stability_wolves = np.divide(1, (dwolves_dt \
                    + 0.000001)).mul(np.where(wolves_pop == 0, 0, 1))
                population_stability_total = ( population_stability_sheep \
                    + population_stability_wolves) / 2
                aggregate_metric = population_stability_total.sum() \
                    / len(population_stability_total)
                aggregate_metrics.append(aggregate_metric)

                ###Done processing simulation results###
                runsDone = runsDone + 1
                if runsStarted < runsNeeded:
                    simulate(workspace, experiment[runsStarted])
                    runsStarted = runsStarted + 1
    for workspace in nl4py.getAllHeadlessWorkspaces():
        workspace.command("stop")
    return aggregate_metrics
```

Next, the parameter names and ranges are read in using **NL4Py** and used to define the problem definition to be passed into **SALib**.

```
ws = nl4py.newNetLogoHeadlessWorkspace()
ws.openModel("models/Wolf Sheep Predation.nlogo")
#Read in the parameters with NL4Py functions and generate a SALib problem
problem = {
  'num_vars': 6,
  'names': ['random-seed'],
  'bounds': [[1, 100000]]
}
problem['names'].extend(ws.getParamNames()[:-2])
problem['bounds'].extend( \
    [item[0::2] for item in ws.getParamRanges()[:-2]])
```

```
#Remove the initial conditions from the problem space.
del problem['bounds'][problem['names'].index("initial-number-wolves")]
problem['names'].remove("initial-number-wolves")
del problem['bounds'][problem['names'].index("initial-number-sheep")]
problem['names'].remove("initial-number-sheep")
```

Now, **SALib** can be used to sample the parameter space and perform sensitivity analysis. We use Saltelli sampling with Sobol sensitivity analysis.:

```
from SALib.sample import saltelli
from SALib.analyze import sobol
firstOrderSobol = np.array([])
totalSobol = np.array([])
sampleSizes = range(500, 1501, 500 )
for sampleSize in sampleSizes:
    nl4py.deleteAllHeadlessWorkspaces()
    param_values_sobol = saltelli.sample(problem, sampleSize)
    Y = np.array(runForParameters(param_values_sobol))
    Si_Sobol = sobol.analyze(problem, Y, print_to_console = True)
    sobol_s1_abs = np.abs(Si_Sobol["S1"])
    S1_and_interactions_sobol = np.append(sobol_s1_abs, \
        (1 - sobol_s1_abs.sum()))
    firstOrderSobol = np.append(firstOrderSobol, S1_and_interactions_sobol)
    totalSobol = np.append(totalSobol, np.array(Si_Sobol["ST"]))
```

Matplotlib was used to plot the resulting first order and total sensitivity indices. For first order sensitivity indices the following Python script was used:

```
from matplotlib import pyplot
labels = np.append(problem['names'], 'Interactions')
fig = pyplot.figure(figsize = [8, 6])
ax = fig.add_subplot(111)
yS1 = firstOrderSobol.reshape(int(firstOrderSobol.shape[0] \
    / len(sampleSizes)), len(sampleSizes), order = 'F')
ax.stackplot(sampleSizes, yS1, labels = labels)
ax.legend(loc = 'upper center', bbox_to_anchor = (0.5, 1.1), \
    ncol = 3, fancybox = True, shadow = True)
pyplot.xticks(fontsize = 18)
pyplot.xlabel("Number of Samples", fontsize = 18)
pyplot.yticks(fontsize = 18)
pyplot.ylabel("S1", fontsize = 18)
pyplot.xticks(sampleSizes)
fig.savefig('output/SobolWSPS1.png')
```
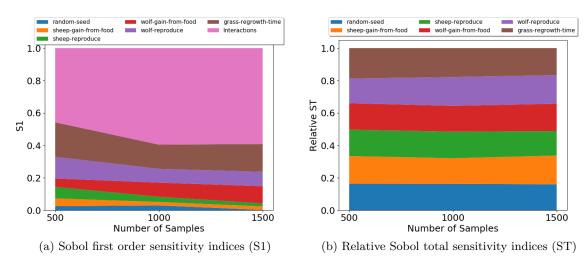
The following was used for total sensitivity indices:

```
yST = totalSobol.reshape(int(totalSobol.shape[0] / len(sampleSizes)), \
    len(sampleSizes), order = 'F')
```

```
for col in range(0, yST.shape[1]):
    yST[:, col] = yST[:, col] / yST[:, col].sum()
fig = pyplot.figure(figsize = [8, 6])
ax = fig.add_subplot(111)
ax.stackplot(sampleSizes, yST, labels = labels)
ax.legend(loc = 'upper center', bbox_to_anchor = (0.5, 1.1), \
    ncol = 3, fancybox = True, shadow = True)
pyplot.xticks(fontsize = 18)
pyplot.xlabel("Number of Samples", fontsize = 18)
pyplot.yticks(fontsize = 18)
pyplot.ylabel("Relative ST", fontsize = 18)
pyplot.xticks(sampleSizes)
fig.savefig('output/SobolWSPST.png')
```


(a) Sobol first order sensitivity indices (S1)   (b) Relative Sobol total sensitivity indices (ST)

Figure 4: Sobol Sensitivity analysis on the Wolf Sheep Predation model under varying sample sizes. Fig. 4a plots the first order sensitivity indices against the uncertainty due to higher-order interactions of the parameters. Relative total sensitivity indices are presented in Fig. 4b by normalizing total sensitivity indices between 0 and 1.

Results of the Sobol sensitivity analysis with Saltelli sampling with varying sample sizes are shown in Fig. 4. Fig. 4a plots the first order sensitivity indices along with the portion of uncertainty due to higher order interactions of parameters, under increasing sample sizes. Fig. 4b compares the total sensitivity indices, normalized between 1 and 0 for comparison, under varying sample sizes.

While the total sensitivity indices demonstrate consistency under increasing sample sizes, higher sample sizes (1000 and 1500) are required for a consistency between first order sensitivity indices. At higher sample sizes, the sensitivity analysis indicates that a majority of the uncertainty of the model is due to higher order interactions between the parameters, an attribute inherent to complex adaptive systems. Parameter `grass-regrowth-time` recorded the highest first order sensitivity, followed by `wolf-reproduce` and `wolf-gain-from-food`. The requirement of a large sample space for consistency in the sensitivity analysis results is indicative of the complexity of the model.

4. Software architecture

NL4Py uses a client-server architecture and consists of two main components, the **NL4Py** client written in Python and the `NetLogoControllerServer` JAR executable written in Java, as shown in Fig. 5. The client code communicates to the `NetLogoControllerServer` through a socket enabled by the **Py4J** library. The entire **NL4Py** package is hosted on the Python package index and is automatically downloaded through the pip installer and sets up the `NetLogoControllerServer` in the **Pip** package installation directory. The client-server architecture allows `NetLogoHeadlessWorkspace`s to be run in parallel as Java threads on the `NetLogoControllerServer`, independent of the user's Python application code. This eliminates the need for users to have to manage the connection to the JVM, thread/process creation, and garbage collection of multiple headless workspaces from their Python application code.

Figure 5: UML Component Diagram of NL4Py

NetLogo provides headless workspaces through its controlling API, which can be controlled through Java or Scala application. NetLogo headless workspaces are inherently thread safe. **NL4Py**, uses this to its advantage by pushing concurrency to the JVM via the `NetLogoContollerServer`. The **NL4Py** Python client provides thread-safe `NetLogoHeadlessWorkspace` objects to the Python application developer, created according to the factory design pattern. Each `NetLogoHeadlessWorkspace` object is mapped to a `HeadlessWorkspaceController` object on the `NetLogoControllerServer`, which is responsible for starting and stopping the NetLogo model, sending commands to the model, fetching results from reporters to the model, querying parameters, and scheduling reporters over model execution. **NL4Py** relieves the Python application of thread/process creation, by ensuring that the procedures with long execution times are non-blocking, i.e., results for procedures such as scheduled reporters, whose results must wait till the end of a model run, return immediately and results can be queried later at a custom time by the user application, without blocking the Python application.

Fig. 6 and Fig. 7 explain the internal organization of the **NL4Py** Python client and `NetLogoControllerServer`, respectively. The **NL4Py** client consists of a
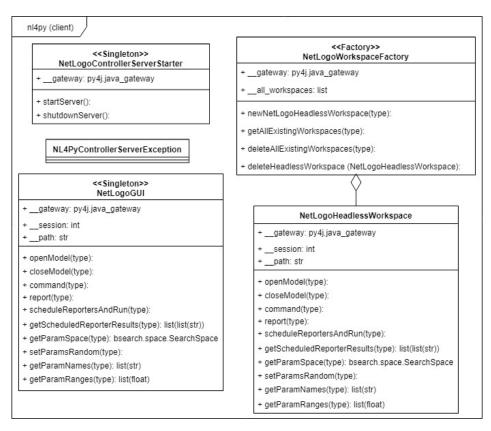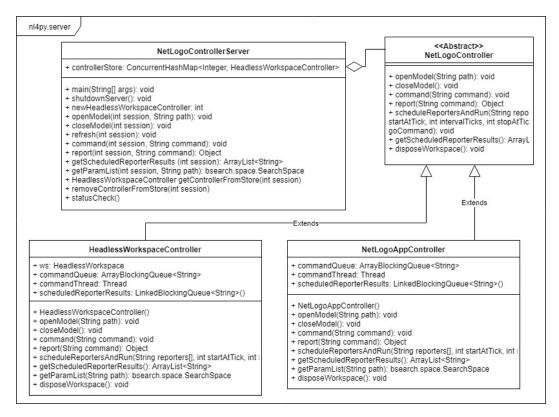
Figure 6: UML Class Diagram of **NL4Py** Python client.

Figure 7: UML Class Diagram of **NL4Py** NetLogoControllerServer.

NetLogoControllerServerStarter object, the NetLogoWorkspaceFactory object, and a NetLogoGUI object respectively. Each object has access to the JVM through the Java_gateway offered by **Py4J**. The NetLogoWorkspaceFactory, is able to create and manage multiple NetLogoHeadlessWorkspace objects, each which maps to a HeadlessWorkspaceController object on the NetLogoControllerServer. On the NetLogoControllerServer, each HeadlessWorkspaceController executes its own command thread for communication with the NetLogoHeadlessWorkspace object through the NetLogo controlling API. Through this command thread, each HeadlessWorkspaceController is able to pass commands and schedule and query from reporters to the NetLogo model concurrently.

5. Performance

This section presents results from measuring execution time improvements made by **NL4Py** on various NetLogo sample models under different configurations and in comparison to execution times by **PyNetLogo**. **NL4Py** is shown to perform faster than both **PyNetLogo** and **PyNetLogo** running in combination with **IPyParallel**.

NL4Py was tested on three NetLogo sample models that accompany the NetLogo model library, namely: the Fire model (Wilensky 1997a), Wolf Sheep Predation model (Wilensky 1997b), and Ethnocentrism model (Wilensky 2003; Axelrod and Hammond 2003). These three models were chosen due to the difference in their stopping conditions:

- Fire: Demonstrates Percolation theory by simulating a forest fire and had a stopping condition for when the fire stopped spreading. This model tends to stop relatively quickly.

- Wolf Sheep Predation: Demonstrates a typical producer vs predator vs prey ecosystem of two animal species and grass, and has two stopping conditions. The model has a wide range of number of ticks before a stop is triggered due to the many possible parameter combinations.

- Ethnocentrism: Demonstrates the emergence of ethnocentric behavior patterns from local competition tendencies that are inherited genetically or culturally. This model has no stopping condition and was stopped after a constant number of ticks had passed.

Each model ran for varying amounts of time due the stopping conditions being triggered earlier or later depending on both the parameter settings and the stochasticity inherent to any ABM. In order to better capture this range of stopping times, each model run was setup with a random parameter initialization. In the case of the Ethnocentrism model, which has no stopping condition, each run was capped at 1000 ticks.

All model runs were performed on an Intel(R) Core i7-7700 CPU (7th generation) with 16GB of memory running Windows 10 (64 Bit). Each experiment was repeated 10 times to account for model stochasticity and all results presented below are aggregates of the multiple runs under the same experimental conditions.

5.1. Profiling over different thread counts

In order to understand the degree to which execution time was improved by parallelizing model runs through **NL4Py**, we studied the time taken to execute the Wolf Sheep Predation model under different numbers of parallelly executing headless workspaces. Specifically, the execution time for 5000, 10000, and 15000 model runs was compared under 1, 4, 8, and 16 parallelly executing `NetLogoHeadlessWorkspace`s. The results are shown in Fig. 8.

It was seen that there is a vast improvement in execution time achieved through parallelizing model runs via **NL4Py**. As the number of model runs were increased for each thread configuration, the execution time also increased linearly. As expected, the optimal number of threads was observed as 8, which was equal to the number of cores on the machine. For higher number of parallel runs, the performance did not improved or even decreased slightly. This can be explained as due to excessive processor context switching, where time is lost when multiple threads must switch between active and inactive states as demand for processor allocations and is higher than the number of available processors.

5.2. Comparison against PyNetLogo

In (Jaxa-Rozen *et al.* 2018), **PyNetLogo** is parallelized with the help of an **IPyParallel** cluster (also referred to as an **IPCluster**). The authors push the `NetLogoLink` objects created by **PyNetLogo** into IPython engines that are able to execute the Python objects linked to NetLogo in parallel. Results of this study find that a combination of **PyNetLogo** and **IPyParallel** is able to execute 14000 runs of the Wolf Sheep Predation NetLogo sample model with randomized parameters within 7161 seconds.
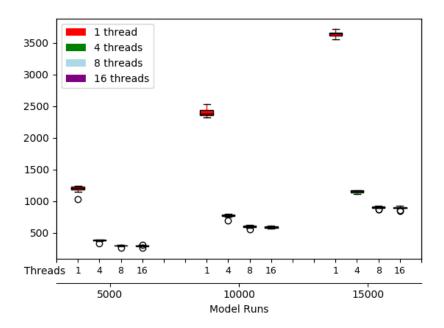
Figure 8: Execution times (in seconds) taken by **NL4Py** to run 1, 4, 8, and 16 parallel **NL4Py** `NetLogoHeadlessWorkspace`s (threads) of the Wolf Sheep Predation sample model provided with `NetLogo` 6.0.2. There is a vast improvement in execution time with parallelization of model runs. However, running more threads than the number of cores available results in no improvement or even slightly slower execution speeds, due to context switching. Runs were performed on an Intel(R) Core i7-7700 CPU (7th generation) with 16GB of RAM running Windows 10 (64 Bit), and repeated 10 times under random parameter sampling.

In order to benchmark the performance of **NL4Py**, we compared its execution times against first, **PyNetLogo** by itself, and second, **PyNetLogo** in unison with **IPyParallel**.

For the comparison against **PyNetLogo**, we considered 200 parallel model runs, i.e., 200 parallely executing headless workspace references from the `Python` script, of each of the three sample models described above. Each model was run for 100, 1000, and 100 ticks for Fire, Ethnocentrism, and Wolf Sheep Predation, respectively. We randomly initialized parameters through `NetLogo` commands, setup, and ran simple reporters for both connectors, checking the end condition for each model at regular intervals, creating a symmetrical `Python` script for both connectors. For example, for the Fire model, the **NL4Py** script was as follows:

```
import time
startTime = int(round(time.time() * 1000))
import nl4py
nl4py.startServer("C:/Program Files/NetLogo 6.0.2")
workspaces = []
modelRuns = 200
for i in range(0, modelRuns):
    n = nl4py.newNetLogoHeadlessWorkspace()
```

```
  n.openModel("./Fire.nlogo")
  n.command("set density random 99")
  n.command("setup")
  n.command("repeat 100 [go]")
  workspaces.append(n)
while len(workspaces) > 0:
    for workspace in workspaces:
    #Check if workspaces are stopped
    ticks = int(float(workspace.report("ticks")))
    stop = str(workspace.report("not any? turtles")).lower()
    if ticks == 100 or stop == "true":
        print(str(modelRuns - len(workspaces)) + " " \
            + str(workspace.report("burned-trees")) + " " \
            + str(ticks) + " " + stop)
        workspaces.remove(workspace)
stopTime = int(round(time.time() * 1000))
totalTime = stopTime - startTime
print(totalTime)
with open("Times_Comparison_Fire.csv", "a") as myfile:
    myfile.write('Fire,' + str(modelRuns) + ',NL4Py,' \
        + str(totalTime) + "\n")
nl4py.stopServer()
```

The corresponding Python script using **PyNetLogo** was:

```
import time
startTime = int(round(time.time() * 1000))
import pyNetLogo
workspaces = []
modelRuns = 200
for i in range(0, modelRuns):
    n = pyNetLogo.NetLogoLink(gui = False,
        netlogo_home = "C:/Program Files/NetLogo 6.0.2",
        netlogo_version = '6')
    n.load_model(r"./Fire.nlogo")
    n.command("set density random 99")
    n.command("setup")
    n.command("repeat 100 [go]")
    workspaces.append(n)
while len(workspaces) > 0:
    for workspace in workspaces:
        #Check if workspaces are stopped
        ticks = int(float(workspace.report("ticks")))
        stop = str(workspace.report("not any? turtles")).lower()
        if ticks == 100 or stop == "true":
            print(str(modelRuns - len(workspaces)) + " " \
                + str(workspace.report("burned-trees")) + " " \
```

```
            + str(ticks) + " " + stop)
        workspaces.remove(workspace)
stopTime = int(round(time.time() * 1000))
totalTime = stopTime - startTime
with open("Times_Comparison_Fire.csv", "a+") as myfile:
    myfile.write('Fire,' + str(modelRuns) + ',PyNetLogo,' \
        + str(totalTime) + '\n')
print(totalTime)
```

Fig. 9 compares the execution times, aggregated over 10 repetitions for each configuration, for simple reporters over 200 model runs for both connectors. It can be seen that **NL4Py** is over twice as fast for Ethnocentrism mode and nearly twice as fast for Fire and Wolf Sheep Predation. Considering that Ethnocentrism ran for more simulation time than the other two models, it can be said that **NL4Py** was able to execute individual NetLogo runs faster than **PyNetLogo**.
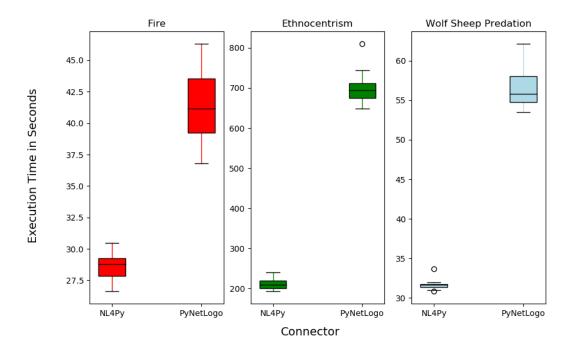

Figure 9: Execution Time Comparison between **NL4Py** vs **PyNetLogo** with simple reporters checking the end of each simulation run, for 200 runs each of three different NetLogo sample models: Ethnocentrism, Fire, and Wolf Sheep Predation. Both controller configurations ran 8 headless workspaces/instances in parallel and executed each sample model for 100, 1000, and 100 ticks, respectively.

5.3. Comparison against PyNetLogo with IPyParallel

Finally, we compared the execution times of the Wolf Sheep Predation model by **NL4Py** against **PyNetLogo** in combination with **IPyParallel** for 5000, 10000, and 15000 model runs.

As the **PyNetLogo** + **IPyParallel** combination requires IPython and **Jupyter** notebook, both connectors were run on **Jupyter** notebooks for fair comparison. The number of parallely executing models was fixed to 8 for both connectors, i.e., 8 threads/ NetLogoHeadlessWorkspaces for **NL4Py** and 8 **IPCluster** engines for the **PyNetLogo** + **IPyParallel** combination. Execution times were aggregated over 5 repetitions for each connector - model runs configuration.

The program running the **NL4Py** execution time measurement for this experiment was written in two functions. The first took in a headless workspace as a parameter, initialized a simulation with random parameter values within the ranges given by the model, and scheduled reporters to run for 100 simulation ticks.

```
import nl4py
# Replace argument to startServer(...) with the location of your NetLogo
#     installation
nl4py.startServer("C:/Program Files/NetLogo 6.0.2")
import time
def simulate(workspace_):
    workspace_.command("stop")
    # Same netlogo commands as used for the PyNetLogo evaluation
    workspace_.command("random-seed 999")
    workspace_.command("set initial-number-sheep 150")
    workspace_.command("set initial-number-wolves 150")
    workspace_.command("set grass-regrowth-time 50")
    workspace_.command("set sheep-gain-from-food 25")
    workspace_.command("set wolf-gain-from-food 50")
    workspace_.command("set sheep-reproduce 10")
    workspace_.command("set wolf-reproduce 10")
    workspace_.command("set show-energy? false")
    workspace_.command('set model-version "sheep-wolves-grass"')
    workspace_.command('setup')
    workspace_.scheduleReportersAndRun( \
        ["ticks", 'count sheep', 'count wolves'], 0, 1, 100, "go")
```

The second, started 8 headless workspaces and used the simulate function to run simulations on these workspaces until the number of runs needed was satisfied.

```
def measureExecutionTime(runsNeeded):
    startTime = int(round(time.time() * 1000))
    runsDone = 0
    runsStarted = 0
    allResults = []
    # Make sure we start 8 headless workspaces to compare to
    # 8 IPCluster engines running PyNetLogo
    for i in range(0, 8):
        workspace = nl4py.newNetLogoHeadlessWorkspace()
        workspace.openModel('./Wolf Sheep Predation.nlogo')
        simulate(workspace)
        runsStarted = runsStarted + 1
```

```
    while (runsDone < runsNeeded):
        for workspace in nl4py.getAllHeadlessWorkspaces():
            newResults = workspace.getScheduledReporterResults()
            # Make sure the models have finished running.
            if len(newResults) > 0:
                allResults.extend(newResults)
                runsDone = runsDone + 1
                if runsStarted < runsNeeded:
                    simulate(workspace)
                    runsStarted = runsStarted + 1
    stopTime = int(round(time.time() * 1000))
    print(stopTime - startTime)
```

We then used these functions to measure the execution times for different numbers of total runs needed under the 8 `NetLogoHeadlessWorkspaces` via **NL4Py**.

```
import os
outputFile = '../output/5.3_output.csv'
out = open (outputFile, "a+")

for runs in [5000, 10000, 15000]:
    executionTime = measureExecutionTime(runs)
    out.write("nl4py,scheduledReporters," + str(runs) + "," \
        + str(executionTime) + "\n")
    nl4py.deleteAllHeadlessWorkspaces()
    out.flush()
```

In order to have the equivalent implementation in **PyNetLogo** running on 8 `IPCluster` engines, using **IPyParallel** (Jaxa-Rozen *et al.* 2018), we first had to, externally, start an `IPCluster`. The first step was to install **IPyParallel** with **pip** tools and start a `IPCluster` of 8 engines. This had to be done through the command prompt, external to **Jupyter** notebook:

```
>pip install ipyparallel
>ipcluster start -n 8
```

Next, the following code was used to connect to the `IPCluster` from the **Jupyter** notebook and push the current working directory into the engines:

```
import numpy as np
import pyNetLogo
import os
import ipyparallel
client = ipyparallel.Client()
print(client.ids)
direct_view = client[:]
import os
engines that can be accessed later
direct_view.push(dict(cwd = os.getcwd()))
```

Following (Jaxa-Rozen *et al.* 2018), we then used the `%%px` decorator to parallelize a **Jupyter** notebook cell to be run by the `IPCluster` engines:

```
%%px #Jupyter notebook decorator
import os
os.chdir(cwd)
import pyNetLogo
import pandas as pd
netlogo = pyNetLogo.NetLogoLink(gui = False, \
    netlogo_home = "C:/Program Files/NetLogo 6.0.2", netlogo_version = '6')
netlogo.load_model('Wolf Sheep Predation.nlogo')
```

Next, a `simulation` function was defined. This function initialized the model on the `IPCluster` to random parameters via **PyNetLogo**, started the simulation run, and repeated reporters over 100 simulation ticks:

```
def simulation(runId):
    try:
        # Same netlogo commands as used for the NL4Py evaluation
        netlogo.command("random-seed 999")
        netlogo.command("set initial-number-sheep 150")
        netlogo.command("set initial-number-wolves 150")
        netlogo.command("set grass-regrowth-time 50")
        netlogo.command("set sheep-gain-from-food 25")
        netlogo.command("set wolf-gain-from-food 50")
        netlogo.command("set sheep-reproduce 10")
        netlogo.command("set wolf-reproduce 10")
        netlogo.command("set show-energy? false")
        netlogo.command('set model-version "sheep-wolves-grass"')
        netlogo.command('setup')
        # Run for 100 ticks and return the number of sheep and wolf agents
        # at each time step
        counts = netlogo.repeat_report(
            ['ticks', 'count sheep', 'count wolves'], 100)
        print(counts)
        results = pd.Series([counts.shape[0],
                            counts.iloc[-1 : 1],
                            counts.iloc[-1 : 2]],
                            index = ['ticks', 'Avg. sheep', 'Avg. wolves'])
        return results
    except Exception as e:
            print(e)
        pass
```

Finally, we could measure the execution times for different total model run counts by **PyNet-Logo** on 8 `IPCluster` engines. For each measurement, we used a `direct_view` from **IPyParallel** to map the `simulation` function to the engines, while also specifying the total number of model runs required:

```
import os
outputFile = '../output/5.3_output.csv'
out = open (outputFile, "a+")

import time
for runs in [5000, 10000, 15000]:
    startTime = int(round(time.time() * 1000))
    # Run the simulations and measure execution time
    try:
        # The following failsafe had to be added as PyNetLogo occasionally
        # fails due to duplicate temporary filenames left from prior runs...
        test = os.listdir(os.getcwd())
        for item in test:
            if item.endswith(".txt"):
                os.remove(os.path.join(os.getcwd(), item))
        # ...end failsafe
        results = pd.DataFrame(direct_view.map_sync(simulation, range(runs)))
    except Exception as e:
        print(e)
        pass
    stopTime = int(round(time.time() * 1000))
    executionTime = stopTime - startTime
    out.write("pyNetLogo,repeatReporters," + str(runs) + "," \
        + str(executionTime) + "\n")
    out.flush()
```

Fig. 10 demonstrates that execution time of the Wolf Sheep Predation model via **NL4Py** was still less when compared to **PyNetLogo** executing on an **IPyParallel** cluster. As execution time increases linearly for both connectors, this difference is more prominent for large numbers of model runs.

6. Limitations

A known limitation of **NL4Py** is that the NetLogo application in GUI enabled mode cannot be run as several separate instances. This is due to the singleton design pattern used by the NetLogo 6 application when accessed through the controlling API, which ensures that only one NetLogo GUI application is created per Java virtual machine. To run multiple instances of the NetLogo application in GUI mode, multiple JVM instances would have to be started and, accordingly, **NL4Py** would have to start multiple instances of the NetLogoControllerServer. Such a feature is not yet supported by **NL4Py**.

7. Conclusion

We introduce **NL4Py** to the agent-based modeling community, a means through which model developers can create and interact with parallelly executable NetLogo models. With **NL4Py** developers can now open, set parameters, send commands and schedule reporters to NetLogo
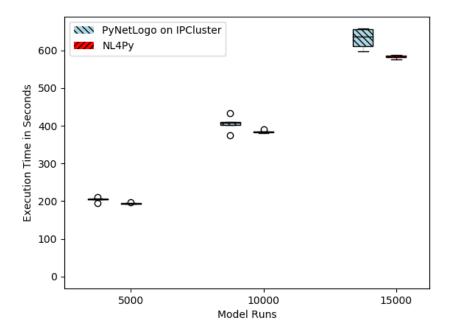
Figure 10: Execution time comparison between **NL4Py** (red) running 8 parallel headless workspaces (8 `Java` threads) vs **PyNetLogo** running on an 8-engine **IPyParallel** `IPCluster` (blue), for the execution of reporters tracking simulation state for each tick over 5000, 10000, and 15000 model runs of the Wolf Sheep Predation model. Parameters are initialized randomly and execution time for each connector - model runs configuration is shown aggregated over 5 repetitions.

models. This is especially useful when performing parameter space exploration on models, where the sheer size of the possible parameter space or mere stochasticity of a model is so great that a vast number of model runs are required to conduct. **NL4Py** is platform independent, being compatible with both `Python 2.7` and `Python 3.6` and has been tested to work on Windows 10, MacOS 10.10, and Ubuntu operating systems.

Sensitivity analysis and parameter calibration of the Wolf Sheep Predation `NetLogo` model were conducted via **NL4Py**, demonstrating the model's sensitivity to higher order interactions of parameters, and the difficulty to achieve a complete equilibrium of both species in the ABM. We demonstrate the ease of conducting these statistical analyses on ABM output using libraries like **SALib** and **DEAP**, once the `NetLogo` model is made controllable through `Python`.

We compare **NL4Py**'s performance on running parallel instances of three `NetLogo` sample models, against the recently introduced `NetLogo` to `Python` connector **PyNetLogo**, and demonstrate an significant improvement in execution time with less effort; without having the user to externally start an `IPCluster` through **IPyParallel**.

Our hope is that this contribution will increase the accessibility of `NetLogo` to researchers from the computational optimization and machine learning domains. The authors see this as a vital link for the ABM community, in order to utilize the latest technological advancements

in easy to use data analytics software packages that are increasingly available on the Python package index.

Acknowledgments

References

Axelrod R, Hammond RA (2003). "The Evolution of Ethnocentric Behavior." In *Midwest Political Science Convention*, volume 2.

Axtell R (2008). "The Rise of Computationally Enabled Economics: Introduction to the Special Issue of the Eastern Economic Journal on Agent-Based Modeling." *Eastern Economic Journal*, **34**(4), 423–428.

Borshchev A (2013). "AnyLogic 7: New Release Presentation." In *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, WSC '13, pp. 4106–4106. IEEE Press, Piscataway, NJ, USA. URL http://dl.acm.org/citation.cfm?id=2675807.2675980.

Chérel G, Cottineau C, Reuillon R (2015). "Beyond Corroboration: Strengthening Model Validation by Looking for Unexpected Patterns." *PloS One*, **10**(9), e0138212.

Collier N, North M (2013). "Parallel Agent-Based Simulation with Repast for High Performance Computing." *Simulation*, **89**(10), 1215–1235.

CoMSES (2018 (Accessed May 29,2018)). *CoMSES: A Growing Collection of Resources for Computational Model-Based Science.* Arizona State University and National Science Foundation. URL https://www.comses.net/.

Cukier R, Fortuin C, Shuler KE, Petschek A, Schaibly J (1973). "Study of the Sensitivity of Coupled Reaction Systems to Uncertainties in Rate Coefficients. I Theory." *The Journal of Chemical Physics*, **59**(8), 3873–3878.

Dagenais B (2009–2015). ***Py4J: A Bridge between* Python *and* Java*.* URL https://www.py4j.org.

Farmer JD, Foley D (2009). "The Economy Needs Agent-Based Modelling." *Nature*, **460**(7256), 685.

Fortin FA, De Rainville FM, Gardner MA, Parizeau M, Gagné C (2012). "**DEAP**: Evolutionary Algorithms Made Easy." *Journal of Machine Learning Research*, **13**, 2171–2175.

Gunaratne C (2018). ***NL4Py** 0.5.0.* URL https://pypi.org/project/NL4Py/.

Gunaratne C, Garibay I (2017). "Alternate Social Theory Discovery using Genetic Programming: Towards Better Understanding the Artificial Anasazi." In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 115–122. ACM.

Herman J, Usher W (2017). "**SALib**: an Open-Source Python Library for Sensitivity Analysis." *The Journal of Open Source Software*, **2**(9).

Hunter JD (2007). "**Matplotlib**: A 2D Graphics Environment." *Computing in Science & Engineering*, **9**(3), 90–95.

Janssen MA, Ostrom E (2006). "Empirically Based, Agent-Based Models." *Ecology and Society*, **11**(2).

Jaxa-Rozen M, Kwakkel JH, *et al.* (2018). "**PyNetLogo**: Linking NetLogo with Python." *Journal of Artificial Societies and Social Simulation*, **21**(2), 1–4.

Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, Kelley K, Hamrick J, Grout J, Corlay S, Ivanov P, Avila D, Abdalla S, Willing C (2016). "**Jupyter** Notebooks – A Publishing Format for Reproducible Computational Workflows." In F Loizides, B Schmidt (eds.), *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pp. 87 – 90. IOS Press.

Lee JS, Filatova T, Ligmann-Zielinska A, Hassani-Mahmooei B, Stonedahl F, Lorscheid I, Voinov A, Polhill JG, Sun Z, Parker DC (2015). "The Complexities of Agent-Based Modeling Output Analysis." *Journal of Artificial Societies and Social Simulation*, **18**(4), 4.

Ligmann-Zielinska A, Kramer DB, Cheruvelil KS, Soranno PA (2014). "Using Uncertainty and Sensitivity Analyses in Socioecological Agent-Based Models to Improve their Analytical Performance and Policy Relevance." *PloS One*, **9**(10), e109779.

Lotka AJ (1926). "Elements of Phyisical Biology." *Science Progress in the Twentieth Century (1919-1933)*, **21**(82), 341–343. ISSN 20594941. URL http://www.jstor.org/stable/43430362.

Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G (2005). "Mason: A Multiagent Simulation Environment." *Simulation*, **81**(7), 517–527.

Masad D (2016). *Py2NetLogo*. URL https://github.com/dmasad/Py2NetLogo.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In S van der Walt, J Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 51 – 56.

McKinney W (2011). "**Pandas**: a Foundational Python Library for Data Analysis and Statistics." *Python for High Performance and Scientific Computing*, pp. 1–9.

Mitchell M (2009). *Complexity: A Guided Tour*. Oxford University Press.

North MJ, Collier NT, Ozik J, Tatara ER, Macal CM, Bragen M, Sydelko P (2013). "Complex Adaptive Systems Modeling with Repast Simphony." *Complex Adaptive Systems Modeling*, **1**(1), 3.

Oliphant TE (2006). *A Guide to **NumPy***, volume 1. Trelgol Publishing USA.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, *et al.* (2011). "**Scikit-learn**: Machine learning in Python." *Journal of Machine Learning Research*, **12**(Oct), 2825–2830.

Pérez F, Granger BE (2007). "**IPython**: a System for Interactive Scientific Computing." *Computing in Science and Engineering*, **9**(3), 21–29. ISSN 1521-9615. doi:10.1109/MCSE.2007.53. URL http://ipython.org.

IPython Development Team T (2018). **IPyParallel**: *Using IPython for Parallel Computing*. URL https://ipyparallel.readthedocs.io/en/latest/.

Python Software Foundation (Accessed 29 May, 2018). *PyPI - the Python Package Index*. URL https://pypi.org/.

Saltelli A (2002). "Making Best Use of Model Evaluations to Compute Sensitivity Indices." *Computer Physics Communications*, **145**(2), 280–297.

Saltelli A, Annoni P, Azzini I, Campolongo F, Ratto M, Tarantola S (2010). "Variance Based Sensitivity Analysis of Model Output. Design and Estimator for the Total Sensitivity Index." *Computer Physics Communications*, **181**(2), 259–270.

Saltelli A, Tarantola S, Chan KS (1999). "A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output." *Technometrics*, **41**(1), 39–56.

Sobol IM (2001). "Global Sensitivity Indices for Nonlinear Mathematical Models and their Monte Carlo Estimates." *Mathematics and Computers in Simulation*, **55**(1-3), 271–280.

Stonedahl F, Wilensky U (2010). **BehaviorSearch** *[Computer Software]*. URL http://www.behaviorsearch.org.

Thiele JC (2014). "R Marries NetLogo: Introduction to the **RNetLogo** Package." *Journal of Statistical Software*, **58**(2), 1–41.

Volterra V (1926). "Fluctuations in the Abundance of a Species Considered Mathematically."

Wilensky U (1997a). *NetLogo Fire Model*. Northwestern University, Evanston, IL. URL http://ccl.northwestern.edu/netlogo/models/Fire.

Wilensky U (1997b). *NetLogo Wolf Sheep Predation Model*.

Wilensky U (2003). *NetLogo Ethnocentrism Model*. Northwestern University, Evanston, IL. URL http://ccl.northwestern.edu/netlogo/models/Ethnocentrism.

Wilensky U, Others (1999). *Netlogo*.

Wilensky U, Rand W (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press.

Affiliation:

Chathika Gunaratne
Complex Adaptive Systems Lab

and
Institute for Simulation and Training
University of Central Florida
E-mail: chathika@knights.ucf.edu

Ivan Garibay
Complex Adaptive Systems Lab
and
Industrial Engineering and Management Systems
University of Central Florida
E-mail: igaribay@ucf.edu