EGYPT
Farmers to Pharoahs

Derrick Diana
Harry Heathcock
Kaedon Jon Williams

September 10, 2019

Abstract

Reimplementing a simulation for Ancient Egyptian Farming Communities in Python with a focus on speed and extensibility, from code originally written in Netlogo, with minor changes for cohesion and ease of use as well as quality of life improvements. Testing showed that there were differences between the original simulation and the reimplementation, even when the reimplementation was run in legacy mode.

1 Introduction

Studying ancient civilisations can be difficult due to incomplete information and an inability to test hypotheses. This is a problem which can be partially solved by simulation, if the simulation can be shown to be accurate. A programme was created to reimplement an older simulation model for ancient Egyptyian farming communities in a new language to allow for extensibility and speed improvements, as well as improving the user experience. Another goal was to test whether the model was resilient to slight changes that would be resultant from a reimplementation.

The simulation itself could provide assistance to researchers attempting to study ancient Egypt, allowing them to view how the conditions that were discovered came about in the first place. This could provide insight as to whether slight changes in initial conditions would create a different environment entirely.

The simulation itself is an agent based model, where households claim and farm fields around the Nile River for the grain which they need to survive. Households are each part of a Settlement, which can contain many households. Households grow and thrive, while others die out.

The design process used followed a few steps. The first step was to read and understand the original source code, collecting required functionality and trying to piece together how the system worked as a whole. Once the system was understood, step two began, in which the basic pieces were implemented into an object-oriented python version. This was split into the simulation and the UI, with the simulation very closely following the original code and the UI using the original as a style guide. Once the basics of the simulation and the UI had been created, a set of available data was agreed on and the two systems were brought together. The UI code instantiated a copy of the simulation code,

pulled the data required for rendering each year and called the function which causes the simulation to continue for another year. From there, the final step was just to continue programming features into the simulation and UI separately until both fulfilled the original requirements.

2 Requirements

2.1 Functional

2.1.1 The simulation should produce graphs to display state

This was achieved through the use of FigureCanvasTkAgg() objects as described in the implementation section. This creates a pair graphs which the user can interact with and choose to display any two of the 11 available graphs at any one time.

2.1.2 The user should be able to change inputs to the system

This was achieved through tkinter scale widgets and check boxes which are also described in the implementation section. Users can slide the scales to change starting conditions within the limits set in the original code.

2.1.3 The simulation should run for a fixed period

This is due to the limited applicability of the model, as the simulation is not meant to run for more than 500 years. This was achieved using a counter, that once passed causes the play and step buttons to be greyed out and unusable, forcing the user to click the reset button rather than pushing the simulation beyond its range of applicability.

2.1.4 A window should display the area in which the simulation is taking place, displaying the settlement and field positions

This was achieved using a tkinter canvas object as described in the implementation section. Logic was written to interpret the state of the simulation and translate that into icons on a grid. The meaning of each of these icons is described in the user manual.

2.1.5 The code base should be easily extensible

2.2 Non-Functional

2.2.1 The system should be able to display at least 1 frame second when running at the highest simulation speed possible

This was easily achieved as the program can display at least 5 frames per second even at the highest settings.

2.2.2 Code should be structured to allow for easy understanding of functionality and allow functions to be easily swapped out (extensibility)

This has been achieved through the majority of functionality which is likely to be changed being placed inside of a single class. This allows for replacing the decision-making of the

agent to be replaced simply by replacing a few functions of a single class, which have documented functionality as described in the technical API manual.

2.2.3 Inputs should be controlled by sliders for each of the initial conditions

This has been achieved through the use of Tkinter Scale widgets to provide input to the program.

2.3 Usability

The following factors were considered when designing the user interface:

2.3.1 Error prevention

In order to prevent errors from occurring, buttons which cannot be clicked at any time are disabled. In addition, since sliders cannot be changed while the program is running they too are disabled (indicated by being greyed out) until the reset button is clicked.

2.3.2 Performance

All of the following statements refer to the program being run on a modern CPU. With the display on, the program runs 1000 years in about 45 seconds if the graph rate is set to its default value of 30. If this value is changed to 100, the time taken to run drops to 25 seconds. If the user only wants the final results of the simulation, the simulation may be run without updating the display. In this case, the program runs about 1000 years in 15 seconds.

2.3.3 Learnability

In order to ensure learnability, a tool tip is constantly displayed on the bottom of the interface to ensure that the user is able to have a live description of what each component does. In addition, a user manual was created and included in this document.

2.4 Use Cases

2.4.1 As an Archaeologist/ general user wanting to run the simulation

I want to be able to change input parameters before running the program to see how the results of the program are affected.
Resolution: To do this, the user may select and drag any of the sliders on the left before the simulation has started running, and on doing so the simulation will reset the simulation with the changed inputs.

2.4.2 As an Archaeologist/ general user wishing to analyse results

I want to save one or more of the displayed graphs for later analysis.
Resolution: To do this, the user may either save individual figures by clicking the save icon under the figure or clicking the Save All Figures button.

2.4.3 As an Archaeologist/ general interested experimentalist

I want to be able to run a large number of simulations, analyse and compare the data from different runs.
Resolution: the inclusion of a Run Multiple button which allows the user to run the entire simulation as many times as the user wishes and the output graphs from each run are each saved in their own folder, within a larger containing outer folder.

2.4.4 As an Archaeologist/ general interested experimentalist

I want to be able to view details about the internal state of the system at any time in order to see how parameters change from one year to the next.
Resolution: the user may pause the simulation and then right click on any block that they wish to see the parameters of. This will cause a pop-up to appear next to the block displaying information about the block.

2.4.5 As an Archaeologist/ researcher wishing to present findings to others

I want to be able to be able to repeat runs and have the exact same starting set up in order to be able to reproduce results to show others and reduce the number of uncontrolled variables in experiments.
Resolution: The user is able to click the text-box "manual-seed-enabled" which allows them to enter a 'seed' which is a specific string of characters. So long as the user enters the same string the next time, the starting positions of the settlements will be identical.

3 Design Overview

The programme was designed such that the only interactions between the simulation itself and the UI are in instantiation, the increase of years and the current state of the simulation. This was done for encapsulation purposes and to ensure the integrity of the simulation at all times. The display is split up into the UI and the rendering of the simulation, this was performed for ease of upkeep, as combining the two would result in a monolithic structure which is difficult to work with and maintain. The basic structure is shown in Figure 1.
Due to this being a reimplementation of work that has already been done, the algorithms and data structures were pre-determined. All changes made to the overall algorithm were due to efficiency concerns, or strange behaviour which was corrected.
Data was stored almost exclusively in lists, as order needed to be randomised each tick to remove possible biases due to ordering of operations being performed on the households.

3.1 Changes

A number of changes in the implementation were made, some due to what seemed to be implementation errors in the original code and others due to functionality not making sense with what was trying to be achieved. Those which affect functionality have been implemented as explained, however when the Legacy Mode setting is enabled, these changes are reverted back to how they were in the original implementation.

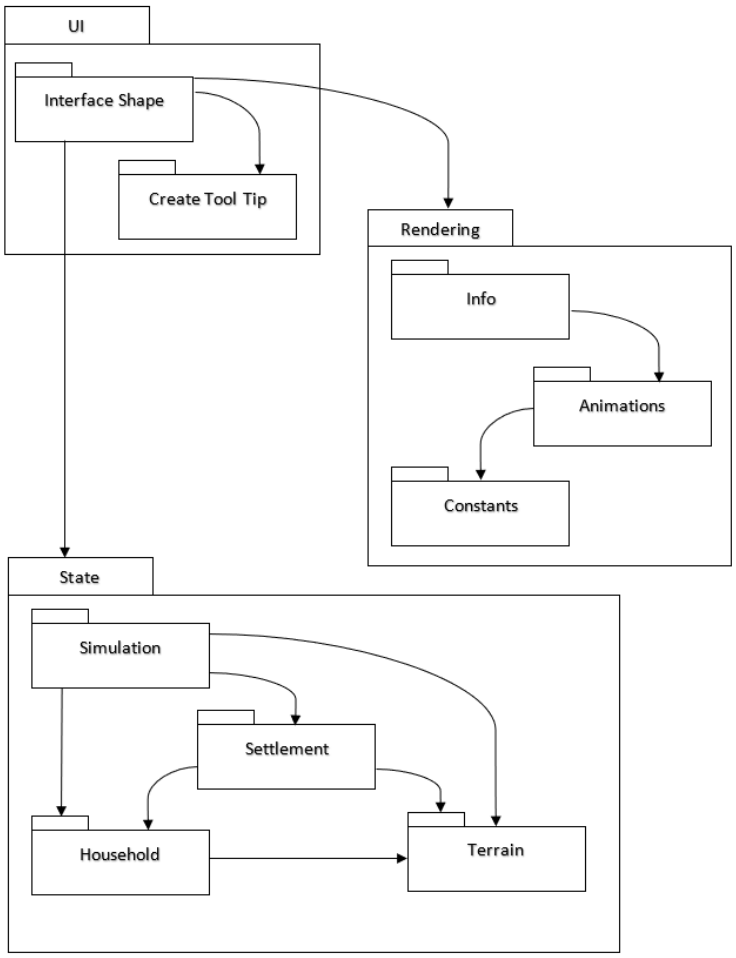

Figure 1: Architecture diagram for the system, showing how the UI checks the state of the simulation and passes this to be rendered.

3.1.1 Death Due to Lack of Grain

In the original NetLogo code, if a household did not have enough grain to survive a year then a single person would die and the grain would be set to zero. In reality this doesn't make sense, as if a household has no food, then no one would survive. The reimplementation changed this so that only the family members who can be fed will survive.

3.1.2 Population Increases

In the original code, if the total current population is less than or equal to the historical projected population then the population can increase. At first glance this seems to be correct, however this allows the population to always grow slightly faster than the projected historical value, and thus in the reimplementation this is changed so that the population will only increase if it is less than the historical projected population.

3.1.3 Generational Changes in Ambition and Competency

In the original code, if the ambition or competency for a household moved out of the bounds set (minimum, up to 1) then a new value would be generated until it fell within the bounds. This means that the computational time for the ambition and competency changes are unbounded. In the reimplementation, a value within the possible range is generated.

3.1.4 Order of Field Claimants

In the original code, households claimed fields in ascending order of how much grain they currently have. This seemed inconsistent with every other operation being independent of grain or ordering, in the reimplementation, this ordering is randomised each year.

4 Implementation

As can be seen in Figure 2, InterfaceShape is the controller class which invokes other methods in all other classes. InterfaceShape contains the code which sets up the GUI, which consists of the components explained below.
A canvas object on which the main simulation grid is drawn. The main simulation grid displays the households, fields and rivers distinctly and clearly. A panel, that consists of slider and check-box objects, which allows the user to initialise parameters for the simulation. Two FigureCanvasTkAgg() objects which act as widgets and each create a matplotlib-style interactive graph. These graphs can be zoomed, scrolled, and can save their displays as images. These are live representations of figure objects within the matplotlib.pyplot library. Several other buttons and sliders which allow the user to interact with the program in minor ways.

4.1 Important Functions in the GUI and Rendering

The GUI has a method called MainLoop, which performs the primary actions corresponding to one year of the simulation which includes calling the following methods, in order:

- Simulation.tick()
  Which causes the simulation class to simulate a single year, generating and setting fertility values for the entire set of land, allowing households to claim, farm and rent land, as well as consuming grain, dying out when not enough food is present, allowing for population growth as well as shifts in the household competency and ambition levels.

- Info.drawGridSimulation()
  Which accesses the simulation object, finding the positions and values for fields and settlements in order to render them on the canvas correctly.

- Info.plotData()
  Which reads the necessary data out of the simulation object and appends all the

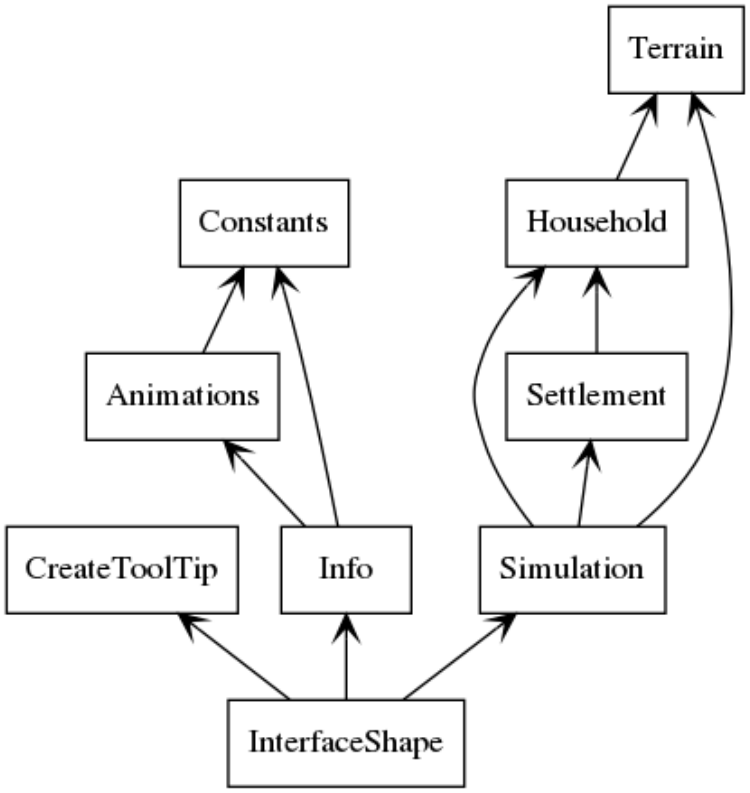

Figure 2: Diagram showing the relationship between all the classes involved in the programme.

data needed for each graph to a list which stores the lists required for each graph. This outer list consists of 11 inner list, each corresponding to a single graph.

- Info.updateGraphs()
  Which takes the data which has already been stored and adds it to the pyplot.figure objects which the FigureCanvasTkAgg() represent. In order to reduce the time taken to draw all the points, only points which have been added since the last draw are drawn each cycle. This is performed until the user decides to change which graph is being displayed, in which case the entire graph must be redrawn, or many lines have already been drawn, in which case the graph is redrawn as a single line. The second part is a speed optimisation, as plotting many points at once is a slow process, however the fragmentation that is created when many smaller lines have been drawn can cause the process to slow even further.

- Info.showGraphs()
  Which calls show on both of the FigureCanvasTkAgg() objects to show the data to the user.

The MainLoop method then calls itself again in a non-recursive manner using the tkinter method after(), which calls the MainLoop function again without blocking after a given delay in milliseconds. This number of milliseconds for the delay is determined by a slider that is present in the GUI and the delay is calculated each frame to achieve the desired frames per second. If the desired framerate cannot be achieved, then the programme does not sleep.

4.2 Important Functions in the Simulation

4.2.1 Simulation

The most important function of simulation is tick(). This calls all other functions which are required for the simulation to run for a single year, incrementing all values which need to be incremented. This begins by calling flood(), which sets fertility values in the land. The settlement tick is then called, which is described below, followed by the rentLand() function. The rentLand() function sorts all the households by ambition and allows ambitious households to farm extra land which they don't own. The settlement tock function is then called, also described below.

4.2.2 Settlement

The two most important functions of the settlement are tick() and tock(). The first consists of functions which should be called before land rental can occur, while the latter consists of the functions which should be called afterwards. The tick() function causes each household within the settlement to claim land, and then farm their claimed land. The tock() function causes each household within the settlement to consume grain for the year, and then allows for a change in ambition and competency, as well as allowing for population growth if other values are also met.

4.2.3 Household

All of the functions in the Household class are important for the functioning of the simulation, as these are where the individual actions are controlled. The farm() function causes the household to farm its fields, the claimLand() function causes the household to make an attempt at claiming fields in the nearby area, grainTick() consuming grain for the year, populationIncrease() giving the possibility of increasing the population and generationalChange() allows the competency and ambition of the household to change over time.

4.2.4 Terrain

The important functions in the Terrain class are the setFertility(), claim() and unclaim() functions. These are all self-explanatory from the names.

5 Test Cases

To validate the correctness of our simulation we ran it using a variety of different initial conditions and then compared the graphs produced to those of the original simulation. The multiple run feature was used extensively to significantly shorten time spent gathering data. The test cases were all run with legacy mode enabled and so included the bugs of the original. Test initial conditions are set to the defaults unless otherwise specified. The following values constitute the defaults:

- Model time span - 500
- Starting settlements - 14

- Starting households - 7
- Starting household size - 5
- Starting grain - 3000
- Min ambition - 0.1
- Min competency - 0.5
- Generational variation - 0.9
- Knowledge radius - 20
- Distance cost - 10
- Fallow limit - 4
- Population growth rate - 0.1
- Min fission chance - 0.7
- Household fission - disabled
- Land rental rate - 50
- Land rental - disabled

The households are divided up into brackets based on their current grain amount in comparison to the household with the most grain. The brackets are $> 66\%$, 33% to 66% and $< 33\%$.

5.1 Test cases considered

5.2 All values are left at the default

This test was chosen to give a baseline set of graphs to compare all subsequent ones to to give an indication as to whether the desired functionality was achieved, relatively speaking. These will also be compared to ones taken from the original simulation to ascertain correctness.

Expected behaviour is for the households in the upper bracket to be the fewest, the middle brackets the second most and the lower bracket the least. Few settlements should have all their population reduced to 0, if any.

5.3 Land rental enabled

This test was chosen to validate the implementation of land rental. The graphs generated shall be compared with the baseline graphs to see if they show the expected differences.

As households descend into poverty and lose fertile fields, they will start to borrow from their neighbours and so bring themselves back from the bad situation they were in, and so fluctuating between poverty and the middle bracket.

The number of households in the lower and middle brackets should fluctuate greatly throughout the simulation, but the number of households in the lower bracket will remain the majority and next will be the households in the middle bracket, though these numbers will be much closer than in the baseline test.

5.4 Min ambition set to 1

This test was chosen to validate the implementation of ambition. The graphs generated shall be compared with the baseline graphs to see if they show the expected differences, as well as with the competency graphs to see if they show similar results.

Since everyone is ambitious, everyone will attempt to farm more and so fewer households will become poor and die out.

The majority of households shall be within the middle bracket, next the lower bracket and then the higher bracket. Though, at times, the lower bracket may overtake the middle bracket or be less than the higher bracket.

5.5 Min competency set to 1

This test was chosen to validate the implementation of ambition. The graphs generated shall be compared with the baseline graphs to see if they show the expected differences, as well as with the ambition graphs to see if they show similar results.

Since everyone is competent, full value is gotten from each harvest and so it should be difficult for any one household to become too poor and die out.

The majority of households shall be within the middle bracket, next the lower bracket and then the higher bracket. Though, at times, the lower bracket may overtake the middle bracket or be less than the higher bracket.

5.6 Min ambition and competency set to 0

This test was chosen to test the other extreme of min ambition and competency. The graphs generated shall be compared with the baseline graphs to see if they show the expected differences.

Households will vary greatly and so there should be a large difference in equality. Since the average household wont be able to farm particularly well and wont do so incredibly often, grain values will be low and a lot of households will die out.

The distribution of households between the brackets should be comparable to that of the baseline test, though at lower values of households in each. The gini index will be relatively high and total grain values will be lower in comparison.

5.7 Household fission enabled

This test was chosen to validate the implementation of land rental. The graphs generated shall be compared with the baseline graphs to see if they show the expected differences.

Since households have a chance to split when they grow large, there will be fewer large households and so fewer households in the higher and middle brackets (since they wont be able to farm as much grain if their workers keep splitting off).

The difference between the graphs of baseline test and this one will not be that different, however, since fission only comes into effect later on in the simulation (since households dont grow to enough workers to split particularly quickly), and so does not have time to greatly affect the bracket distribution. The one graph that is affected is the number of households and settlements graph, in which the number of households begins to increase towards the end of the simulation.

6 Discussion of Results

6.1 All values are left at the default

As can be seen, the graphs look quite different. In the graphs from our reimplementation the household numbers drop off very quickly, which does not correctly represent the original NetLogo simulation. The cause of this problem is not known, but possible theories

include the claim land functionality stopping certain household from claiming enough fields to sustain themselves, as well as there being an error in the harvest calculation. While the Households holding stated percentage of wealthiest households grain graphs do look quite different, they do show the same behaviour of how inequality is distributed between the households when taking into account the error with our households dying off.

So overall a good representation of how inequality is distributed, but not a good representation of how households grow and die off.

6.2 Land rental enabled

Interestingly enough, with rent enabled in our simulation it performs the same or possibly worse than our baseline test. Which is due to negative values being harvested a good amount of the time, which impacts the grain of the households and so causes more to die off. This does not follow expected behaviour and so is an error.

7 Conclusions

The application as designed seems to run much faster than its predecessor, while also being written in a more extensible manner. For clients, there are quality of life features that were implemented which were not in the original, for example being able to set up multiple runs of the same settings to automatically continue as well as being able to easily save graphs which are generated. In the original, a single graph could be saved at a time. In the reimplementation, all graphs could be saved at once, or a graph could be zoomed in on before being saved.

References

A Manual

A.1 Sliders

The left hand side of the interface contains a set of sliders which may be adjusted to give different parameters to the program. On adjustment, the sliders automatically reset the simulation with the changed values. Below sliders are check boxes, which function similarly - resetting the simulation on being changed. One check box asks the users for a seed and then passes this to the simulation.

A.2 Simulation display

The central portion of the interface displays the current state of the simulation, in which the user may right click any block to see a description of the current parameters associated with that block. To clear the display of information when no longer required, the user may left click. The information displayed depends on what type of block was clicked on, as follows:

A.2.1 Settlement

Settlements are represented by houses, where the size of the house represents the total population of the settlement.
When right clicked on, settlements display their total population, and how many households are contained within.

A.2.2 Fields

Fields are represented by a circle (unharvested) or a piece of barley. The colour of the symbol is related to the colour of the household which has claimed the field, with purple representing the top 33% of wealth, blue representing the middle 33% and yellow representing the bottom 33%.
When right clicked on, fields will display whether or not they have been harvested, their fertility, their potential harvest for the year and the number of years it has been since the field has been harvested.

A.2.3 Land

The shade of green of each of the block of represents its fertility, where a darker colour means that it is more fertile. Right clicking on a piece of land displays its fertility and what its potential harvest would be if it were to be claimed and harvested.

A.3 Graphs of data

Two displays are available on the right hand side of the interface. Each is controlled by a drop down box beneath it which when clicked allows the user to choose one of the eleven graphs to be displayed. There is also a tool bar underneath each graph which allows the user to interact with the graph through actions such as panning or zooming. To save an individual graph one may click the save button in the toolbar. To save all the graphs into a folder of your choosing, click Save all figures and enter the name of the folder.

B Technical API Manual

This section will assume a familiarity with how the simulation is meant to function as well as the code of Simulation, Settlement, Household and Terrain, although a brief description of their functionality will be described. If additional variables are required in Settlement, Household or Terrain, note that these variables must first be added to the "__slots__" list at the top of the class in order to be used. This is to provide speedup to accessing variables and reduce memory usage overall and is opposed to the usual python dictionary which is used to store variables within classes.

B.1 Simulation

The Simulation class forms a wrapped for all of the other simulation objects, holding data that is required globally throughout. It also controls some of the order of operations. For extensibility, the simulation class would probably not provide much help outside of

changes to instantiation. If one were to want different households or settlements to run with different parameters, this is where one would make the changes.

B.2 Settlement

The Settlement class holds the households which exist in a specific area together, controls the removal of households when their population reaches 0 and ensures that the households within run their yearly simulations in a random order. If one were to want to change the order in which the simulation is run, or change behaviour of household removal, this is the file in which changes should be made.

B.3 Household

The Household class controls the year to year activities of individual houses, including farming, using food, population increases and changes to ambition and competency over generations. If one were to want to change any general behaviour of the individual households, this is likely to be the correct area to look. Due to how python treats functions, functions can be replaced in certain households without affecting how all households function. If this is done, it must be ensured that the population of the settlement and total population are adjusted when the population is adjusted, and that the grain of the household at the end of the year is added to the total grain.

B.4 Terrain

The Terrain class maintains information about the state of the land in a specific position, as well as containing information about what its position is. It is unlikely that extensibility would come from altering this class, but if it is necessary, then it must be ensured that the years not harvested is incremented each year, harvested is set to false at the start of the year, the fertility and harvest are set every year, and the claim and unclaim functions need to maintain their current functionality.

C Screenshots

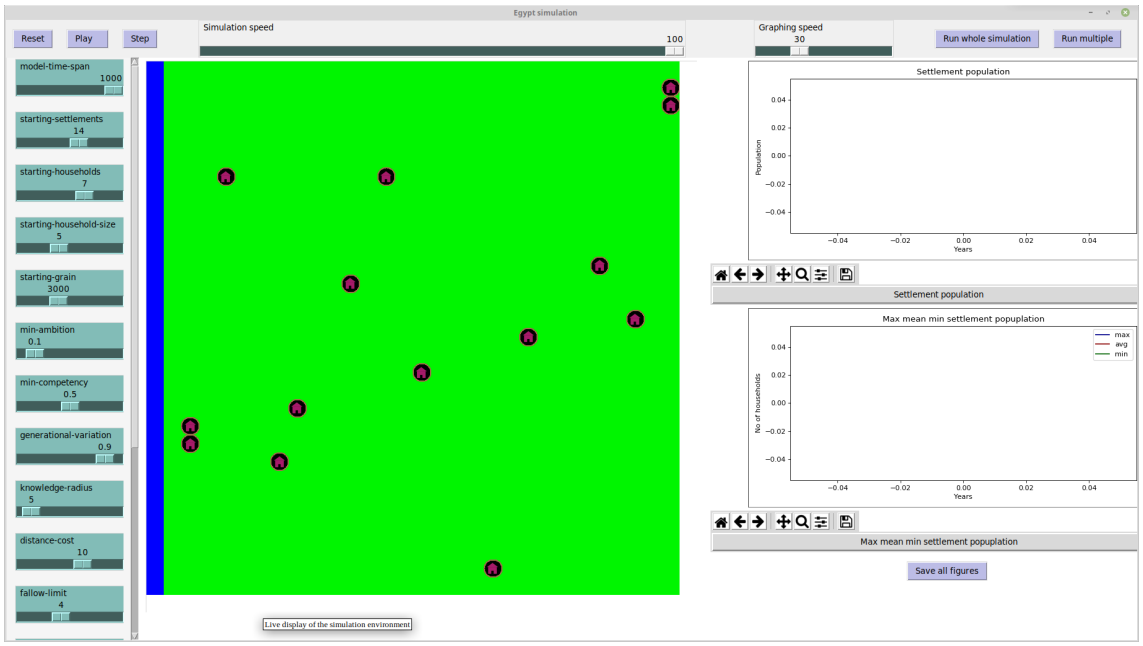

Figure 3: Screenshot of the application once opened, without any actions being performed.

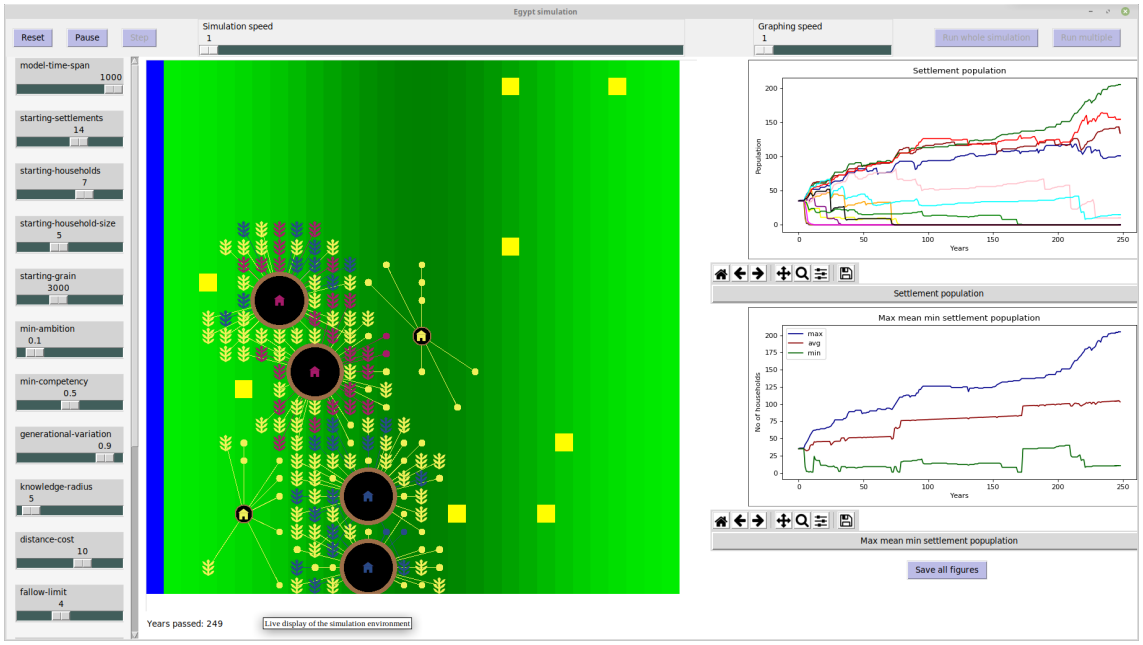

Figure 4: Screenshot of the application once a run has begun, taken at a slow simulation speed to prevent screentearing during a screenshot.

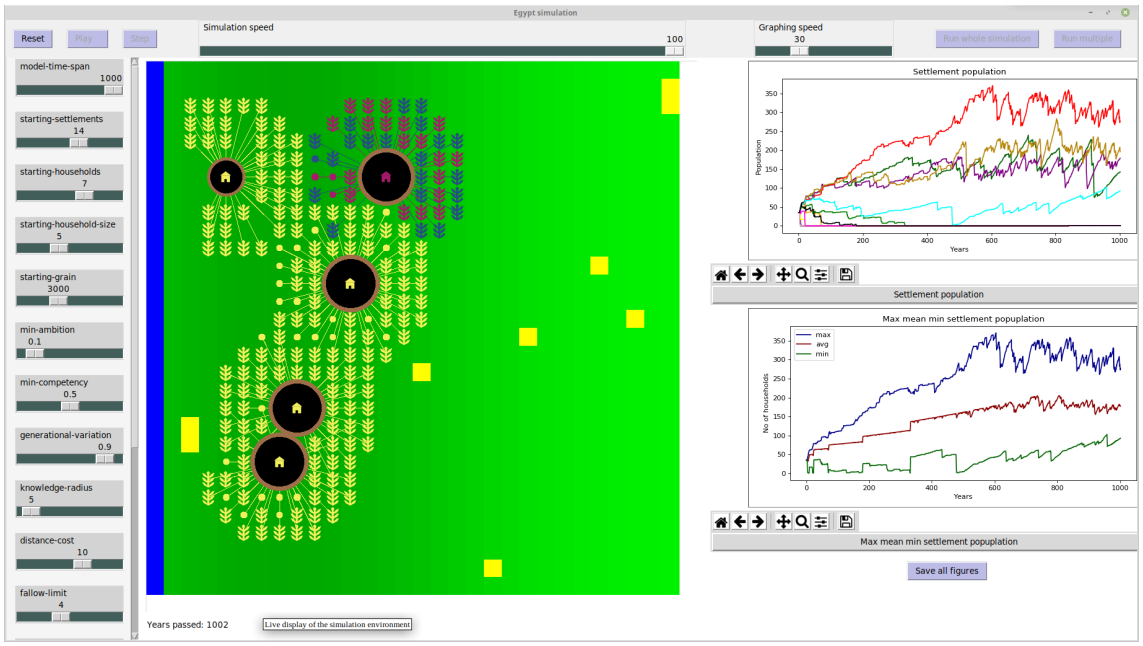

Figure 5: Screenshot of the application once a run has completed.

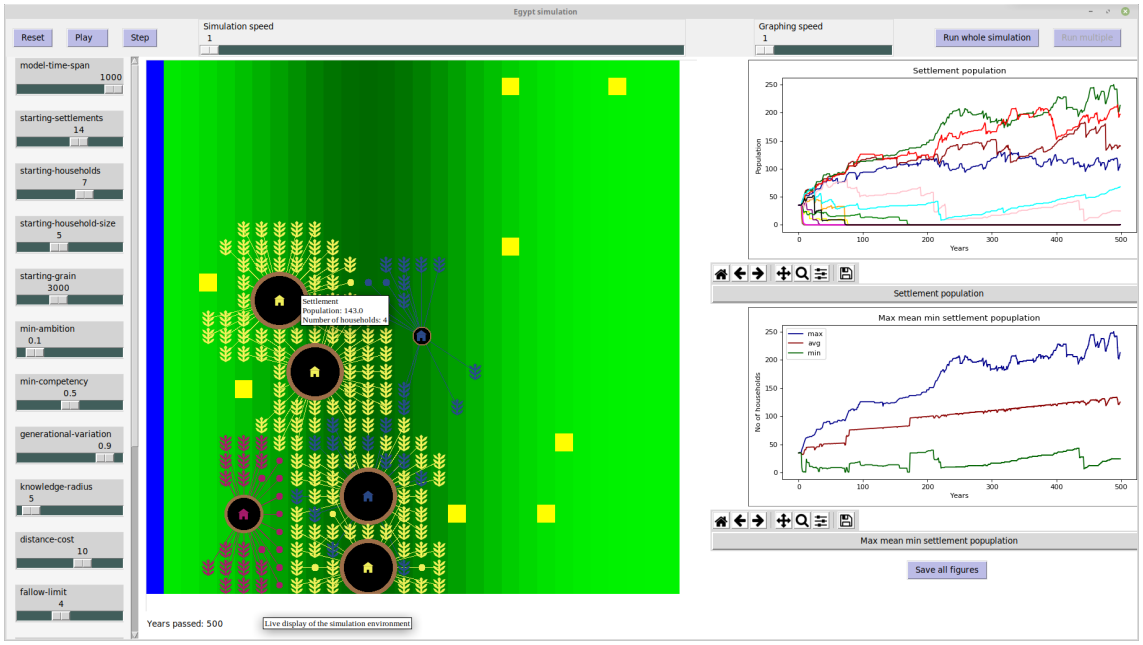

Figure 6: Screenshot of the application during a run, with the simulation paused and the output of right clicking on a settlement shown.

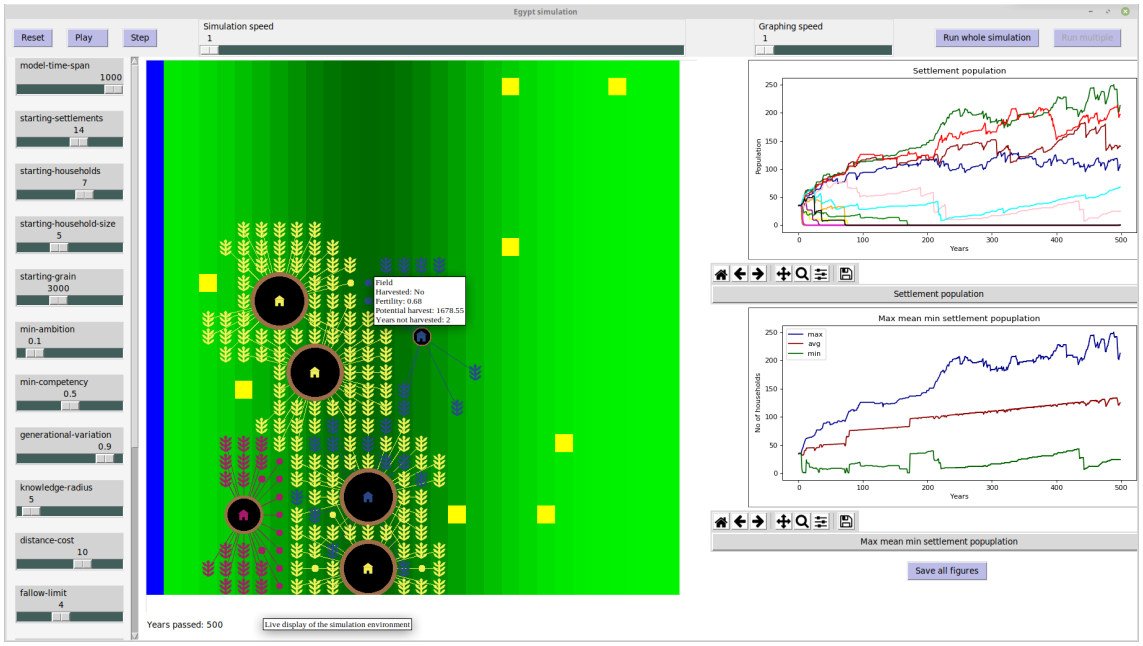

Figure 7: Screenshot of the application during a run, with the simulation paused and the output of right clicking on an unharvested field shown.

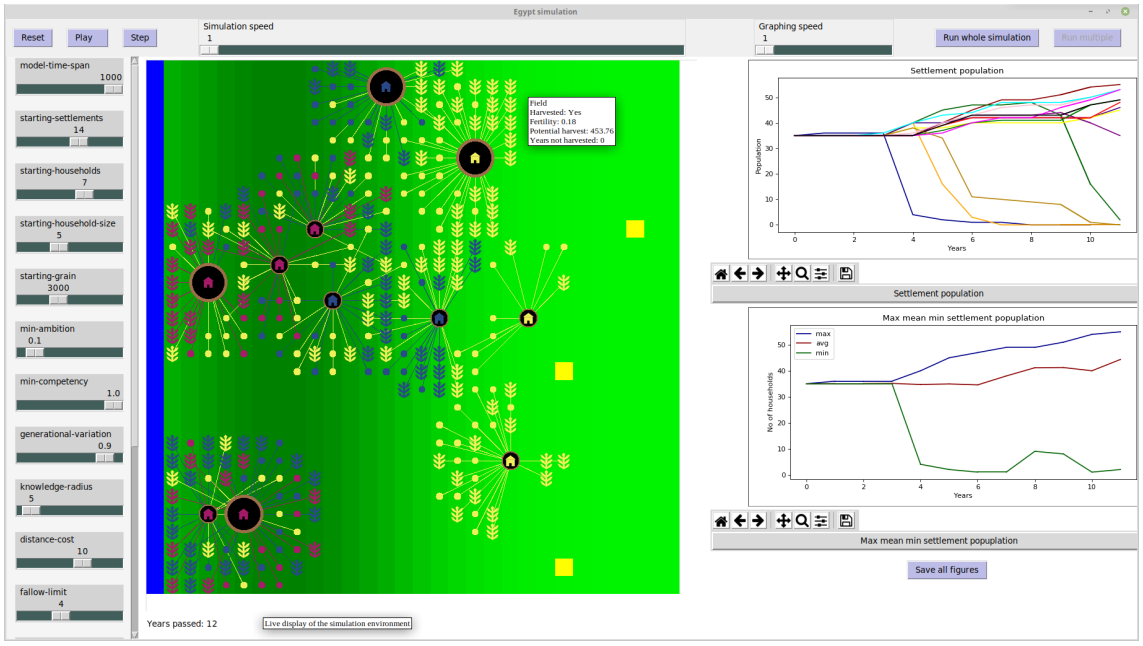

Figure 8: Screenshot of the application during a run, with the simulation paused and the output of right clicking on a harvested field shown.

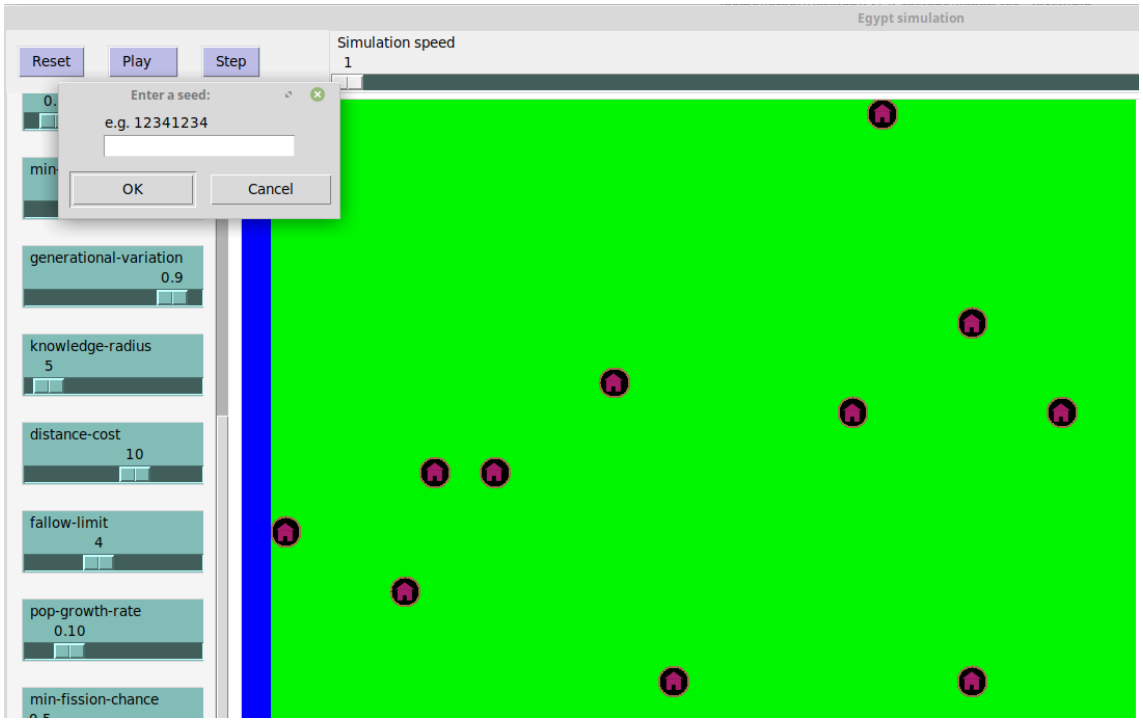

Figure 9: Screenshot of the application showing how a manual seed is entered.

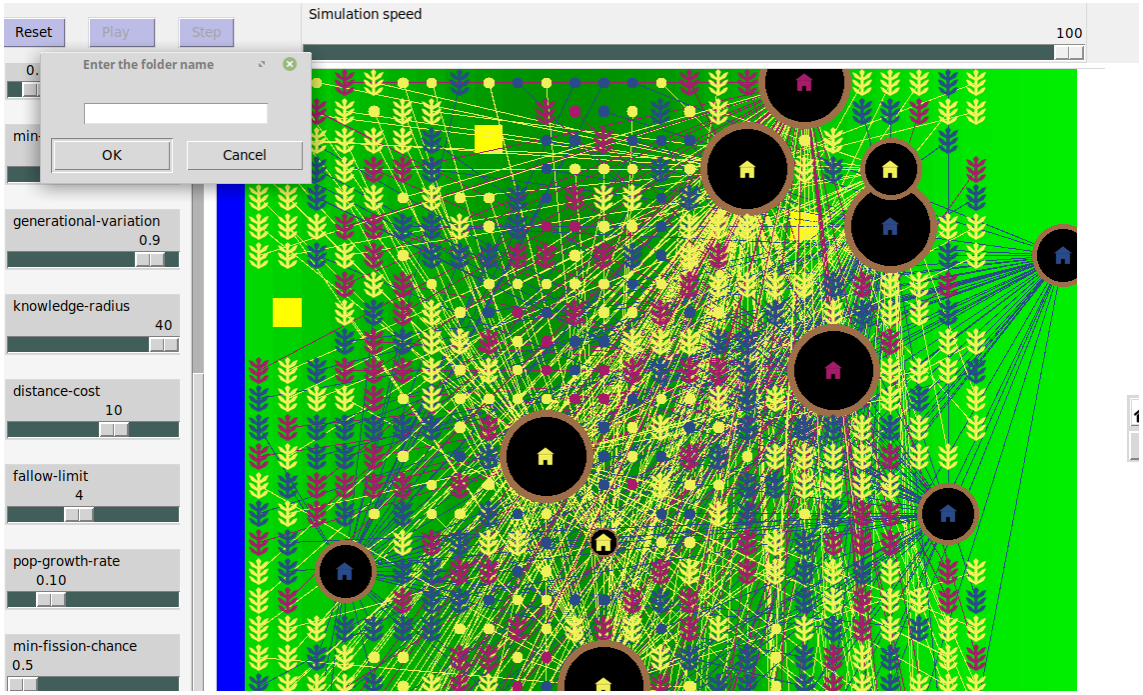

Figure 10: Screenshot of the application showing the dialog asking for a foldername to save multiple figures in for the "Save all Figures" button.

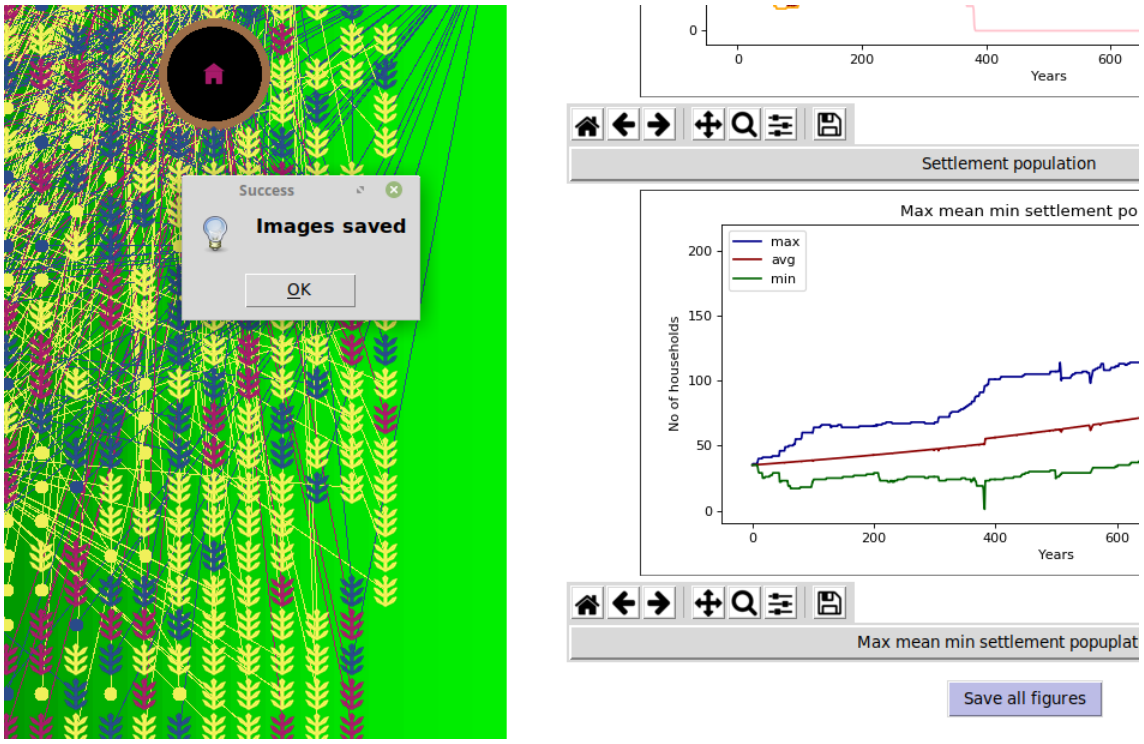

Figure 11: Screenshot of the application showing output of the "Save all Figures" button once all figures have been saved.