geonomics Documentation

Release 1.1.4

Drew Hart

Jun 11, 2021
# Getting Started

| 1 | Description | 3 |
| 2 | Key Features | 5 |
| 3 | About the Documentation | 7 |
| 4 | Indices and tables | 101 |
|  | Python Module Index | 103 |
|  | Index | 105 |
A Python package for simulation of genomic evolution on complex landscapes. Provides for easy construction of individual-based, spatially explicit, forward-time simulation models under arbitrarily complex scenarios.

Using minimal code, build models with arbitrarily complex scenarios, including spatially varying selection, selection on multiple, monogenic or polygenic traits, and non-stationary demographic and environmental change.
CHAPTER 2

Key Features

1. An object-oriented scripting framework, allowing for easy model construction, customization, and extension
2. Model set-up from a single, well annotated parameters file
3. Tools for customizable collection of data throughout a simulation
4. Ability to model complex evolutionary scenarios, including custom demographic change, spatially varying selection, and multiple polygenic traits
5. Ability to model complex spatial scenarios, including multi-layer simulated or real-world landscapes, resistance-based movement, and non-stationary environmental change
6. Numerous visualization tools, to help the user design models and explore results
About the Documentation

This documentation is designed to be read from the top down, as information becomes increasingly detailed. To jump right in, check out Getting started and Examples.

For more information, see Motivation, Data structures, and Operations.

For fine detail about particular Parameters, Data structures, or Operations, see those sections.

Merry modeling!

### 3.1 Installation

Some Geonomics dependencies (i.e. other Python packages that Geonomics uses) depend in turn on a stack of open-source geospatial libraries in C (GEOS, GDAL, and PROJ).

They can sometimes be tricky to install. Please make sure you have them installed first. Then you can follow the pip installation instructions below.

#### 3.1.1 Installing with pip

If you already have GEOS, GDAL, and PROJ installed, then installing with pip probably will not be an issue.

The pip software can be installed by following the installation instructions on the pip webpage.

Then, Geonomics can be installed using:

```
pip install geonomics
```

The installation can then be tested by launching a Python prompt and running:

```
import geonomics as gnx
gnx.run_default_model()
```
This will load the Geonomics package, create in your current working directory a new Geonomics parameters file containing the default parameter values, use that file to instantiate and run a Geonomics model, then delete the parameters file (by default).

### 3.1.2 Dependencies

**Required dependencies:**

- numpy
- matplotlib
- pandas (version 0.23.4 or later)
- geopandas
- scipy (version 1.3.1 or later)
- scikit-learn
- statsmodels (version 0.9.0 or later)
- shapely
- bitarray
- rasterio

**Optional dependencies:**

- nlmyp

---

### 3.1.3 Troubleshooting

Here is a list of issues that we have seen come up during Geonomics installation, each with some suggestions for troubleshooting:

**Mac Segmentation Fault**

Geonomics installs, but running the default model crashes Python with a segmentation fault:

We have only seen this issue on older Macs (ca. 2012-2015). The segmentation fault appears to be caused by functions within scipy’s interpolate module (e.g. `scipy.interpolate.griddata`), such as discussed here). The only fix we are aware of, as of now, is to use conda to set up a clean `conda` environment, then install Geonomics and its dependencies there. While we are not sure why, this appears to make things all better.

---

### 3.2 Getting started

#### 3.2.1 Quick start

For the beginner, we recommend the following steps:
1. Review the diagram of a typical workflow, below.

2. Read the ‘Overview’ subsection (below), and the following two sections (‘Data structures’ and ‘Operations’), to get a general understanding of the logic, components, and necessary and optional behaviors of a Geonomics model.

3. Skim the ‘Parameters’ section, to understand the structure and use of a Geonomics parameters file.

4. Use pip to install Geonomics ($ pip install geonomics);

5. Open Python and run import geonomics as gnx;

6. Use the gnx.make_parameters_file function, to begin creating template parameters files that they can tweak as desired;

7. Use the gnx.make_model function and then the Model.walk or Model.run methods to instantiate and run the models they’ve parameterized;

8. Use the various Model.plot methods to visualize the behavior and results of their models.

As you work through these steps, it may be helpful to refer to the Workflow Diagram for guidance.

### 3.2.2 Really Quick Start

For the impatient beginner:

1. Install Geonomics
2. Launch Python
3. Import Geonomics

```bash
>>> import geonomics as gnx
```

4. Run the default model, and start mussin’ around!

```bash
>>> mod = gnx.run_default_model()
```

This will load the Geonomics package as gnx, create a default Geonomics parameters file in your current working directory, then use that file to instantiate and run a Model using the default parameter values.

### 3.3 Workflow Diagram

The following is a graphical depiction of a typical Geonomics workflow. You may want to reference this as you are working through the steps outlined in the Quick start.
1.) Create and edit a parameters file

```json
parameters:

landscape:
  -

community:
  
  species:
    
    genomic_arch:
      -
        }

model:
  -
```

Optional: Include paths to raster files and/or directories of environmental-change rasters

- **raster file:**
- **raster directory:**

Optional: include path to a genomic architecture file (.csv)

```text
genomic architecture file:
loc, p, dom, r, trait, alpha,
1, 0.5, 0.5, 0.5,
1, 0.5, 0.5, 0.5,
1, 0.5, 0.01, 0.5, 0.5,
1, 0.5, 0.01, 1.0, 0.5,
1, 0.5, 0.01, 2.0, 0.5,
1, 0.5, 0.5, 0.5,
1, 0.5, 0.5, 0.5,
```

2.) Use the parameters file to create a model

- **Landscape**
  - Characteristics:
    - dimensions
    - number of layers
    - layers, rasters
    - environmental change events

- **Species**
  - Characteristics:
    - carrying capacity
    - layer
    - fitness
    - history
    - parameters
    - genome
    - architecture
    - demographic
    - change events

- **Individual**
  - Characteristics:
    - genome
    - 0.0
    - 1.0
    - 0.1
    - phenotype

3.) Run the model

- **Model**
  - Characteristics:
    - number of runs
    - steps per run
    - statistics to calculate
    - data to collect

- **Output**
  - for n runs:
    - for t timesteps per run:
      - movement
      - mating
      - write output
      - mortality
      - change events

```text
output directory:
```
3.4 Motivation

Backward-time (i.e. coalescent) simulators abound. But they are inadequate for simulation of many scenarios of interest, including: natural selection on traits with arbitrary genomic architectures; spatially variable natural selection; simulation of species or populations distributed continuously and moving realistically across complex landscapes; complex demographic change simultaneous with ongoing, often non-stationary environmental change; and coevolutionary interactions between multiple species or incipient species. Few existing forward-time simulators can model all of these phenomena, and those that can are incredibly powerful, but often impose a high cost of entry. Geonomics aims to fill this empty niche by combining ease of use with broad extensibility. If it succeeds at doing this, Geonomics should prove uniquely useful for a wide range of purposes, from intro-level educational use to high-quality theoretical, methodological, empirical, and applied research.

Geonomics is written in Python, a full-fledged scripting language that is relatively easy to learn (and fun!). In Python, it can be pretty quick for a new user to get up to speed and start doing useful work. For work with Geonomics, this turnaround time should be even quicker. Geonomics aims to require minimal Python knowledge (yet maintain high extensibility for interested, expert users). Essentially, anyone should be able to build their own, arbitrarily complex Geonomics models as long as they know how to install the package, open a Python console, call Python functions, and edit some default values in a pre-packaged script.

3.5 Examples

STILL UNDER CONSTRUCTION!

3.6 Conceptual Diagram

As you read through the Overview, you may want to refer to this conceptual diagram as a useful reference.

The primary Data structures are depicted in the central image.

Most of the key Operations are depicted in the cycle of four corner boxes.
3.7 Overview

As you’re reading, you may want to refer to the Conceptual Diagram as a useful reference. The primary data structures are depicted in the central image. Most of the key operations are depicted in the cycle of four corner boxes.
3.7.1 Data structures

The following sections discuss the structure and function of the key Geonomics classes. Users will interface with these classes more or less directly when running Geonomics models, so a fundamental understanding of how they’re organized and how they work will be useful.

Landscape and Layer objects

One of the core components of a Geonomics model is the land. The land is modeled by the Landscape class. This class is an integer-keyed dict composed of numerous instances of the class Layer. Each Layer represents a separate environmental variable (or ‘layer’, in GIS terminology), which is modeled a 2d Numpy array (or raster; in attribute ‘rast’), of identical dimensions to each other Layer in the Landscape object, and with the values of its environmental variable ‘e’ constrained to the interval \([0 <= e <= 1]\). Each Layer can be initialized from its own parameters subsection within the ‘land’ parameters section of a Geonomics parameters file.

For each Species (see Individual, Species, and Community objects, below), the different Layer layers in the Landscape can be used to model habitat viability, habitat connectivity, or variables imposing spatially varying natural selection. Landscape and Layer objects also contain some metatdata (as public attributes), including the resolution (attribute ‘res’), upper-left corner (‘ulc’), and projection (‘prj’), which default to 1, (0,0), and None but will be set otherwise if some or all of the Layer layers are read in from real-world GIS rasters.

Genomes, GenomicArchitecture, and Trait objects

Individual objects (see Individual, Species, and Community objects, below) can optionally be assigned genomes. If they are, each Individual’s genome is conceptually modeled as a 2-by-L array (where 2 is the ploidy, currently fixed at diploidy, and L is the genome length) containing 0s and 1s (because Geonomics strictly models biallelic SNPs, i.e SNPs with ‘0’- and ‘1’-alleles). (In actuality, Geonomics stores genomes by combining Numpy arrays for non-neutral genotypes with a tskit TableCollection for neutral genotypes and for the current population’s spatial pedigree. Although this makes for more complicated data structures, it optimizes information retention while minimizing memory usage, keeping Geonomics fast yet nonetheless enabling powerful spatiotemporal population genomics research. See tskit.tables.TableCollection, below, for details.)

The parameter L, as well as numerous other genomic parameters (including locus-wise starting frequencies of the 1 alleles; locus-wise dominance effects; locus-wise recombination rates; and genome-wide mutation rates for neutral, globally deleterious, and adaptive loci), are controlled by the GenomicArchitecture object pertaining to the Species to which an Individual belongs. (For the full and detailed list of attributes in a GenomicArchitecture object, see its class documentation, below.) The genomes of the initial Individuals in a simulation are drawn, and those of Individuals in subsequent generations are recombined (and optionally mutated) according to the values stipulated by the GenomicArchitecture of their Species. The user can create a species with a GenomicArchitecture and with corresponding genomes by including a ‘genome’ subsection in that species’ section of the Geonomics parameters file (and setting the section’s various parameters to their desired values).

Geonomics can model Individuals’ phenotypes. It does this by allowing the user to create an arbitrary number of distinct Traits for each Species. Each trait is represented by a Trait object, which maps genomic loci onto that trait, maps effect sizes (‘alpha’) onto those loci, and sets the trait’s polygenic selection coefficient (‘phi’). An Individual’s phenotype for a given trait is calculated as the ‘null phenotype’ plus a weighted sum of the products of its ‘effective genotypes’ at all loci underlying that Trait and the effect sizes (i.e. ‘alpha’)] of those loci:

\[
z_{i,t} = \text{null\_genotype} + \sum_{l=0}^{n} \alpha_{t,l} g_{i,l}
\]
where $z_{i,t}$ is the phenotype of Individual $i$ for trait $t$, $g_{i,l}$ is the effective genotype of Individual $i$ at locus $l$, and $\alpha_{t,l}$ is the effect size of locus $l$ for trait $t$.

The ‘null phenotype’ represents the phenotypic value for an Individual who is homozygous for the 0 allele at all loci for a trait. For monogenic traits the null phenotype is 0 and the effect size is fixed at 0.5 (such that individuals can have phenotypes of 0, 0.5, or 1); for polygenic traits the null phenotype is 0.5 and effect sizes can be fixed at or distributed around a mean value (which is controlled in the parameters file).

The ‘effective genotype’ refers to how the genotype is calculated based on the dominance at a locus, as indicated by the following table of genotypes:

<table>
<thead>
<tr>
<th>Biallelic genotype</th>
<th>Codominant</th>
<th>Dominant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 : 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 : 1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>1 : 1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(For the full and detailed list of attributes in a Trait object, see its class documentation, below.)

Note that for maximal control over the GenomicArchitecture of a Species, the user can set the value of the ‘gen_arch_file’ parameter in the parameters file to the name of a separate CSV file stipulating the locus numbers, starting 1-allele frequencies, dominance effects, traits, and inter-locus recombination rates (as columns) of all loci (rows) in the GenomicArchitecture; these values will override any other values provided in the ‘genome’ subsection of the species’ parameters.

### Individual, Species, and Community objects

Being that Geonomics is an individual-based model, individuals serve as the fundamental units (or agents) of all simulations. They are represented by objects of the Individual class. Each Individual has an index (saved as attribute ‘idx’), a sex (attribute ‘sex’), an age (attribute ‘age’), an x,y position (in continuous space; attributes ‘x’ and ‘y’), and a list of environment values (attribute ‘e’), extracted from the Individual’s current cell on each Layer of the Landscape on which the Individual lives.

The Species class is an OrderedDict (defined by the collections package) containing all Individuals, (with their ‘idx’ attributes as keys). If a Species has a GenomicArchitecture then the Individuals in the Species will also each have genomes (attribute ‘g’), and the GenomicArchitecture includes Traits then each individual will also have a list of phenotype values (one per Trait; attribute ‘z’) and a single fitness value (attribute ‘fit’). (These attributes all otherwise default to None.)

Each Species also has a number of other attributes of interest. Some of these are universal (i.e. they are created regardless of the parameterization of the Model to which a Species inheres). These include: the Species’ name (attribute ‘name’); its current density raster (a Numpy array attribute called ‘N’); and the number of births, number of deaths, and terminal population size (i.e. total number of individuals in the Species) of each timestep (which are list attributes called ‘n_births’, ‘n_deaths’, and ‘Nt’). If the Species was parameterized with a GenomicArchitecture then that will be created as the ‘gen_arch’ attribute (otherwise this attribute will be None).

All of the Species in a Model are collected in the Model’s Community object. The Community class is simply an integer-keyed dict of Species. For the time being, the Community object allows a Geonomics Model to simulate multiple Species simultaneously on the same Landscape, but otherwise affords no additional functionality of interest. However, its implementation will facilitate the potential future development of methods for interaction between Species. (e.g. to simulate coevolutionary, speciation, or hybridization scenarios).
Model Objects

Objects of the Model class serve as the main interface between the user and the Geonomics program. (While it is certainly possible for a user to work directly with the Landscape and Species or Community objects to script their own custom models, the typical user should find that the Model object allows them accomplish their goals with minimal toil.) The main affordance of a Model object is the Model.run method, which, as one could guess, will run the Model. The typical workflow for creating and running a Model object is as follows:

1. Create a template paramters file containing the desired sections, by calling gnx.make_parameters_file with all relevant arguments;
2. Define the scenario to be simulated, by opening and editing that parameters file (and optionally, creating/editing corresponding files, e.g. genomic-architecture CSV files; or raster or numpay-array files to be used as Layers);
3. Instantiate a Model object from that parameters file, by calling mod = gnx.make_model('/path/to/params_filename.py')
4. Run the Model, by calling mod.run().

For detailed information on usage of these functions, see their docstrings. When a Model is run, it will:

1. Run the burn-in (until the minimal burn-in length stipulated in the parameters file and the built-in stationarity statistics determine that the burn-in is complete);
2. Run the main model for the stipulated number of timesteps;
3. Repeat this for the stipulated number of iterations (retaining or refreshing the first run’s initial Landscape and Species objects as stipulated).

The Model object offers one other method, however, Model.walk, which allows the user to run a model, in either ‘burn’ or ‘main’ mode, for an arbitrary number of timesteps within a single iteration (see its docstring for details). This is particularly useful for running Geonomics within an interactive Python session. Thus, Model.run is primarily designed for passively running numerous iterations of a Model, to generate data for analysis, whereas Model.walk is primarily designed for the purposes of learning, teaching, or debugging the package, or developing, exploring, introspecting, or visualizing particular Models.

Secondary (i.e. private) classes

The typical user will not need to access or interact with the following classes in any way. They will, however, parameterize them in the parameters file by either leaving or altering their default values. Geonomics sets generally sensible default parameter values wherever possible, but for some scenarios they may not be adequate, and for some parameters (e.g. the window-width used by the _DensityGridStack; see below), there is no “one-size-fits-most” option. Thus, it is important that the user have a basic acquaintance with the purpose and operation of these classes.

_ConductanceSurface

The _ConductanceSurface class allows Geonomics to model a Species' realistic movement across a spatially varying landscape. It does this by creating an array of circular probability distributions (i.e. VonMises distributions), one for each cell on the Landscape, from which Individuals choose their directions each time they move. To create the _ConductanceSurface for a Species, the user must indicate the Layer that should be used to create it (i.e. the Layer that represents landscape permeability for that Species). The _ConductanceSurface's distributions can be simple (i.e. unimodal), such that the maximum value of the distribution at each cell will point toward the maximum value in the 8-cell neighborhood; this works best for permeability Layers with shallow, monotonic gradients, because the differences between permeability values of neighboring cells can be minor (e.g. a gradient representing the directionality of a prevalent current). Alternatively, the distributions can be mixture (i.e. multimodal)
distributions, which are weighted sums of 8 unimodal distributions, one for each neighboring cell, where the weights are the relative cell permeabilities (i.e. the relative probabilities that an Individual would move into each of the 8 neighboring cells); this works best for non-monotonic, complex permeability Layers (e.g. a DEM of a mountainous region that is used as a permeability Layer). (The Landscape is surrounded by a margin of 0-permeability cells before the _ConductanceSurface is calculated, such that Landscape edges are treated as barriers to movement.) The class consists principally of a 3d Numpy array (y by x by z, where y and x (a.k.a i and j, or latitude and longitude) are the dimensions of the Landscape and z is the length of the vector of values used to approximate the distributions in each cell.

_DensityGridStack_

The _DensityGridStack class implements an algorithm for rapid estimating an array of the local density of a Species. The resulting array has a spatial resolution equivalent to that of the Landscape, and is used in all density-dependent operations (i.e. for controlling population dynamics). The density is estimated using a sliding window approach, with the window-width determining the neighborhood size of the estimate (thus essentially behaving like a smoothing parameter on the density raster that is estimated, with larger window widths producing smoother, more homogeneous rasters). The window width can be controlled by setting the ‘density_grid_window_width’ parameter in the ‘mortality’ section of the Species parameters, in a parameters file; however, if the default value (None) is left then the window width will default to 1/20th of the width of the Landscape. Note that setting the window width to a value less than ~1/20th of the Landscape width is likely to result in dramatic increases in runtime, so this is generally advised against (but may be necessary, depending on the user’s interests). The following plot show the estimated density rasters for a 1000x1000-cell Landscape with a population of 50,000 individuals, using various window widths:

Density estimations for various window_widthvalues
1000x1000-cell landscape with 50000 individuals

![Density estimations for various window_widthvalues](image)

3.7. Overview
And this plot shows how \_DensityGridStack creation (plot titled ‘making’) and runtime (plot titled ‘calc’) scale with window-width for that Landscape:

\begin{center}
\includegraphics[width=\textwidth]{plot.png}
\end{center}

\_KDTree

The _KDTree class is just a wrapper around scipy.spatial.cKDTree. It provides an optimized algorithm (the kd-tree) for finding neighboring points within a given search radius. This class is used for all neighbor-searching operations (e.g. mate-search).

\tskit.tables.TableCollection

To enable easy recording of the pedigree of a simulated Species, Geonomics depends on the Python package tskit (software that originated as improvements made to the data structures and algorithms used by the popular coalescent simulator msprime). Geonomics uses the tskit tables API to store the full history of individuals, genotypes, mating events, and recombinations for a Species in a TableCollection object. This data structure is initiated with a random pedigree that is backwards-time simulated using msprime and used as a stand-in (viz. meaningless) pedigree for a Species' starting population. It is then updated with each timestep’s forward-time simulation information, and it is periodically simplified as recommended by the tskit authors using tskit’s simplification algorithm. (The simplification interval can be parameterized by the user.) Because each individual is stored along with its x,y birth location, a TableCollection thus contains the full spatial pedigree of a Species' current population. Geonomics additionally provides some wrapper functions, implemented as Species methods, for converting the TableCollection to a TreeSequence, and for calculating statistics and creating visualizations from these two
data structures. (For further details regarding tskit, see the Python package documentation and the associated peer-reviewed paper.)

_RecombinationPaths

The _RecombinationPaths class contains a large (and customizable) number of bitarrays, each of which indicates the genome-length diploid chromatid numbers (0 or 1) for a recombinant gamete produced by an Individual of a given Species (henceforth referred to as ‘recombination paths’). These recombination paths are generated using the genome-wide recombination rates specified by the Species’ GeonomicArchitecture. They are generated during construction of the Model, then drawn randomly as needed (i.e. each time an Individual produces a gamete). This provides a reasonable trade-off between realistic modeling of recombination and runtime.

_LandscapeChanger and _SpeciesChanger

These classes manage all of the landscape changes and demographic changes that were parameterized for the Landscape and Species objects to which they inhere. The functions creating these changes are defined at the outset, then queued and called at their scheduled timesteps.

_DataCollector and _StatsCollector

These classes manage all of the data and statistics that should be collected and written to file for the Model object to which they inhere (as determined by the parameters file used to create the Model). The types of data to be collected, or statistics to be calculated, as well as the timesteps at which and methods by which they’re collected/calculated and determined at the outset, then the appropriate functions called at the appropriate timesteps.

3.7.2 Operations

The following sections discuss the mechanics of core Geonomics operations.

Movement and Dispersal

Movement is optional, such that turning off movement will allow the user to simulate sessile organisms (which will reproduce and disperse, but not move after dispersal; this distinction is of course irrelevant for a Species with a maximum age of 1). For Species with movement, Individuals can move by two distinct mechanisms. Spatially random movement is the default behavior; in this case, Individuals move to next locations that are determined by a random distance drawn from a Wald distribution and a random direction drawn from a uniform circular (i.e. Von Mises) distribution. As with most distributions used in Geonomics, the parameters of these distributions have sensible default values but can be customized in a Model’s parameters file (see section ‘Parameters’, below).

The alternative movement mechanism that is available is movement across a permeability surface, using a _ConductanceSurface object. To parameterize a _MovementSurface for a Species, the user must create a template parameters file that includes the necessary parameters section for the Species (i.e. the user must set ‘movement’ to True and ‘movement_surface’ to True in the Species’ arguments to the gnx.make_parameters_file function (see the docstring for that function for details and an example). Individuals move to next locations determined by a random distance drawn from a Wald distribution and a random direction drawn from the distribution at the _ConductanceSurface cell in which which the Individuals are currently located. For details about _ConductanceSurface creation, see section ‘_ConductanceSurface’ above, or the class’ docstring.
Dispersal is currently implemented identically to spatially random movement (with the caveat that the an offspring’s new location is determined relative its parents’ midpoint). But the option to use a _ConductanceSurface for dispersal will be offered soon.

Reproduction

Each timestep, for each Species, potential mating pairs are chosen from among all pairs of individuals within a certain distance of each other (i.e. the mating radius, which is set in the parameters file). This choice can be made by strict nearest-neighbor mating, or pairs can be randomly drawn from within the mating radius using either uniform or inverse-distance weighted probabilities. These pairs are subsetted if necessary (i.e. if the Species requires that Individuals be above a certain reproductive age, or that they be of opposite sexes, in order to mate; these values can also be changed from their defaults in the parameters file). Remaining pairs mate probabilistically (according to a Bernoulli random draw with probability equal to the Species’ birth rate, which is also set in the parameters file).

Pairs that are chosen to mate will produce a number of new offspring drawn from a Poisson distribution (with lambda set in the parameters file). For each offspring, sex is chosen probabilistically (a Bernoulli random draw with probability equal to the Species’ sex ratio), age set to 0, and location chosen by dispersal from the parents’ midpoint (see section ‘Movement and Dispersal’). For Species that have genomes, offspring genomes will be a fusion of two recombinant genomes from each of the two parents (where each recombinant is indexed out a parent’s genome using a recombination path; see section ‘_RecombinationPaths’). For Species with Traits in their GenomicArchitectures, offspring phenotypes are determined at birth. Mutations are also drawn and introduced at this point (see section ‘Mutation for details’).

Mortality

Mortality can occur as a combination of two factors: density dependence and natural selection. Each Individual has a death decision drawn as a Bernoulli random variable with \( P(d_i) = 1 - P(s_{i\text{den}})P(s_{i\text{fit}}) \), where \( P(d_i) \) is the probability of death of Individual \( i \), and \( P(s_{i\text{den}}) \) and \( P(s_{i\text{fit}}) \) are the probabilities of survival of Individual \( i \) given density-dependence and fitness. The probability of density-dependent death is contingent on an Individual’s x,y location (i.e. the cell in which they’re currently located. And an Individual’s probability of survival due to fitness is just equal to the product of their absolute fitness (\( \omega \)) for each of the Individual’s \( m \) Traits. Thus the equation for an Individual’s probability of death becomes:

\[
P(d_i) = 1 - (1 - P(d_{x,y})) \prod_{p=1}^{m} \omega_{i,p}
\]

The following two sections explain in detail the implementation and calculation of the two halves of the right side of this equation.

Density dependence

Density dependence is implemented using a spatialized form of the class logistic growth equation (\( \frac{dN_{x,y}}{dt} = rN_{x,y}(1 - \frac{N_{x,y}}{K_{x,y}}) \), where the x,y subscripts refer to values for a given cell on the Landscape). Each Species has a carrying-capacity raster (a 2d Numpy array; attribute ‘K’), which is defined in the parameters file to be one of the Layers in the Landscape. The comparison between this raster and the population-density raster calculated at each timestep serves as the basis for the spatialized logistic growth equation, because both equations can be calculated cell-wise for the entire extent of the Landscape (using the Species’ intrinsic growth rate, the attribute ‘R’, which is set in the parameters file).
The logistic equation returns an array of instantaneous population growth rates within each cell. We can derive from this the density-dependent probability of death at each cell by subtracting an array of the expected number of births at each cell, then dividing by the array of population density:

\[ P(d_{x,y}) = \frac{E[N_{d;x,y}]/N_{x,y} = \frac{E[N_{b;x,y}] - \frac{dN_{x,y}}{dt}}{N_{x,y}} \]

The expected number of births at each cell is calculated as a density raster of the number of successful mating pairs, multiplied by the expected number of births per pair (i.e. the expectation of the Poisson distribution of the number of offspring per mating pair, which is just the distribution’s parameter lambda).

**Selection**

Selection on a Trait can exhibit three regimes: **spatially divergent**, universal, and spatially contingent. Spatially divergent selection is the default behavior, and the most commonly used; in this form of selection, an Individual’s fitness depends on the absolute difference between the Individual’s phenotypic value and the environmental value of the relevant Layer (i.e. the Layer that represents the environmental variable acting as the selective force) in the cell where the Individual is located.

Universal selection (which can be toggled using the ‘univ_adv’ parameter with a Trait’s section in the parameters file) occurs when a phenotype of 1 is optimal everywhere on the Landscape. In other words, it represents directional selection on an entire Species, regardless of Individuals’ spatial contexts. (Note that this can be thought of as operating the same as spatially divergent selection, but with the environmental variable driving natural selection being represented by an array in which all cells are equal to 1.)

Under spatially contingent selection, the selection coefficient of a Trait varies across space, such that the strength of selection is environmentally determined in some way. Importantly, this selection regime is not mutually exclusive with the other two; in other words, selection on a certain Trait be both spatially contingent and either spatially divergent or universal. Spatially contingent selection can be implemented by providing an array of values (equal in dimensions to the Landscape) to the ‘phi’ value of a Trait, rather than a scalar value (which could be done within the parameters file itself, but may be more easily accomplished as a step between reading in a parameters file and instantiating a Model object from it). (Note that non-spatially cotingent selection could in fact be thought of as a special case of spatially contingent selection, but where the array of selection-coefficients has the same value at each cell.)

All possible combinations of the three selection regimes of selection can all be thought of as special cases of the following equation for the fitness of Individual i for Trait p (\(omega_{i,p}\)):

\[ \omega_{i,p} = 1 - \phi_{p;x,y}(\mid e_{p;x,y} - z_{i;p} \mid)^{\gamma_p} \]

where

\(\phi_{p;x,y}\) is the selection coefficient of trait p; \(e_{p;x,y}\) is the environmental variable of the relevant Layer at Individual i’s x,y location (which can also be thought of as the Individual’s optimal phenotype); \(z_{i;p}\) is Individual i’s (actual) phenotype for Trait p; and \(\gamma_p\) controls the curvature of the fitness function (i.e. how fitness decreases as the absolute difference between an Individual’s optimal and actual phenotypes increases; the default value of 1 causes fitness to decrease linearly around the optimal phenotypic value).

Importantly, most individuals will experience selection on a given trait that is only a fraction of the strength dictated by a trait’s selection coefficient ($\phi_{p;x,y}$). This is because an individual’s fitness for a given trait is determined by the product of the trait’s selection coefficient and the individual’s degree of mismatch to its local environment, ($\mid e_{p;x,y} - z_{i;p} \mid$ in the previous equation), such that only individuals who are extremely mismatched (e.g. an individual of phenotype 1 or greater who is located in a 0-valued environmental cell) will experience selection equal to or exceeding $\phi_{p;x,y}$. Because of this, selection coefficients that would be considered ‘strong’ in classical, aspatial population genetics models will tend to behave less strongly in Geonomics models.
Mutation

Geonomics can model mutations of three different types: neutral, deleterious, and trait mutations. These terms don’t map precisely onto the traditional population-genetic lingo of “neutral”, “deleterious”, and “beneficial”, but they are more or less analogous:

- **Neutral** mutations are the same conceptually in Geonomics as they are in the field of population genetics in general: They are mutations that have no effect on the fitness of the individuals in which they occur.
- **Deleterious** mutations in Geonomics are also conceptually the same in Geonomics and in population genetics: They negatively impact the fitness of the individuals in which they occur.
- **Trait** mutations are the place where the Geonomics concept and the population-genetic concept diverge: In Geonomics, natural selection acts on the phenotype, not the genotype (although these concepts are identical if a Trait in monogenic), and it is (by default, but not always; see section ‘Selection’, above) divergent. For this reason it would be a misnomer to call mutations that influence a given Trait’s phenotypes ‘beneficial’ – even though that term is the closest population-genetic concept to this concept as it is employed in Geonomics – because the same mutant genotype in the same Individual could have opposite effects on that Individual’s fitness in different environmental contexts (i.e. it could behave as a beneficial mutation in one region of the Landscape but as a deleterious mutation in another).

Species interactions

This functionality is not yet included available. But the Community class was created in advance recognition that this functionality could be desirable for future versions (e.g. to simulate coevolutionary, speciation, or hybridization scenarios).

Landscape and Species change

For a given Layer, any number of change events can be planned. In the parameters file, for each event, the user stipulates the initial timestep; the final timestep; the end raster (i.e. the array of the Layer that will exist after the event is complete, defined using the end_rast parameter); and the interval at which intermediate changes will occur. When the Model is created, the stepped series of intermediate Layers (and _ConductanceSurface objects, if the Layer that is changing serves as the basis for a _ConductanceSurface for any Species) will be created and queued, so that they will swap out accordingly at the appropriate timesteps.

For a given Species, any number of demographic change events can also be planned. In the parameters file, for each event, the user stipulates the type of the event ('monotonic', 'cyclical', 'random', or 'custom') as well as the values of a number of associated parameters (precisely which parameters depend on the type of event chosen). As with Landscape change events, all necessary stepwise changes will be planned and queued when the Model is created, and will be executed at the appropriate timesteps.

It is also possible to schedule any number of instantaneous changes to some of the life-history parameters of a Species (e.g. birth rate; the lambda parameter of the Poisson distribution determining the number of offspring of mating events). This functionality is currently minimalistic, but will be more facilitated in future versions.

3.8 Visualization

Each Model object has a variety of visualization methods (Model.plot, Model.plot_fitness, etc.), which aim to help users design, run, explore, present, and explain their Models’ behavior and results. These methods
can be called at any time (e.g. as soon as the Model has been created, or after it has run for any number of timesteps); but it is worth mentioning that some methods may be invalid depending on the point in model-time at which they’re called (e.g. `Model.plot_genotype`, `Model.plot_phenotype`, and `Model.plot_fitness` cannot be run for Models that have not yet been burned in, as they will not yet have genomes assigned), or on the Species for which they’re called (e.g. the aforementioned methods cannot create plots for a Species that has no GenomicArchitecture; and likewise, the `Species.plot_demographic_changes` method cannot be called for a Species for which demographic changes were not parameterized).

Below is a list of the visualization methods available, with example output for each (generated from the default Geonomics Model):

**UNDER CONSTRUCTION!**

### 3.9 Parameters

**How to Use This Section:**

In order to create and run a Geonomics Model, you will need a valid Geonomics parameters file. No worry though – this is very easy to create! To generate a new, template parameters file, you will simply call the `gnx.make_parameters_file` function, feeding it the appropriate arguments (to indicate how many Species and Layers you want to include in your Model; which parameters sections you want included in the file, both for those Layers and Species and for other components of the Model; and the path and filename for your new parameters file). Geonomics will then automatically create the file for you, arranged as you requested and saved where you requested.

When you then open that file, you will see the following:

```python
#<your_filename>.py

#This is a default parameters file generated by Geonomics
#(by the gnx.params.make_parameters_file() function).

params = {

(continues on next page)
```
This is the beginning of a file that is really just a long but simple Python script (hence the `.py` extension); this whole file just defines a single, long, nested `dict` (i.e. a Python ‘dictionary’) containing all of your parameter values. It may look like a lot, but don’t be concerned! For two reasons:

1. All the hard work is already done for you. You’ll just need to change the default values where and how you want to, to set up your particular simulation scenario.

2. You will probably leave a good number of the parameters defined in this file untouched. Geonomics does its best to set sensible default values for all its parameters. Though of course, you’ll want to think clearly nonetheless about whether the default value for each parameter is satisfactory for your purposes.

Each parameter in the parameters value is preceded by a terse comment, to remind you what the parameter does. But for detailed information about each parameter, you’ll want to refer to the following information. What follows is a list of all of the Geonomics parameters (in the sections and the top-to-bottom order in which they’ll appear in your parameters files). For each parameter, you will see a section with the following information:

- a snippet of the context (i.e. lines of Python code) in which it appears in a parameters file;
- the valid Python data type(s) the parameter can take
- the default value of the parameter
- a ranking score, indicating how likely it is that you will want to reset this parameter (i.e. change it from its default value), and encoded as follows:
  - ‘Y’: almost certainly, or must be reset for your Model to run
  - ‘P’: it is quite possible that you will want to reset this parameter, but this will depend on your use and scenario
  - ‘N’: almost certainly not, or no need to reset because it should be set intelligently anyhow (Note: this does not mean that you cannot reset the parameter! if that is the case for any value then it does not appear in the parameters file)
- other relevant, detailed information about the parameter, including an explanation of what it defines, how its value is used, where to look for additional information about parameters related to other Python packages, etcetera

These section will be formatted as follows:

```python
<brief comment about the parameter>
'<param_name>': <default_param_value>,
```
This section should serve as your primary point of reference if you confront any uncertainty while creating your own parameters files. We’ll start with the section of parameters that pertains to the *Landscape* object.

### 3.9.1 Landscape parameters

#### Main

**dim**

```python
# x,y (a.k.a. j,i) dimensions of the Landscape
'dim': (20,20),
```

tuple
default: (20,20)
reset: P

This defines the y,x dimensions of the *Landscape*, in units of cells. As you might imagine, these values are used for a wide variety of basic operations throughout Geonomics. Change the default value to the dimensions of the landscape you wish to simulate on.

**res**

```python
# x,y resolution of the Landscape
'res': (1,1),
```

tuple
default: (1,1)
reset: N

This defines the *Landscape* resolution (or cell-size) in the y,x dimensions (matching the convention of the **dim** parameter). This information is only used if GIS rasters of *Landscape* layers are to be written out as GIS raster files (as parameterized in the ‘Data’ parameters). Defaults to the meaningless value (1,1), and this value generally needn’t be changed in your parameters file, because it will be automatically updated to the resolution of any GIS rasters that are read in for use as *Layers* (assuming they all share the same resolution; otherwise, an Error is thrown).

**ulc**

```python
# x,y coords of upper-left corner of the Landscape
'ulc': (0,0),
```

tuple
default: (0,0)
reset: N
This defines the x,y upper-left corner (ULC) of the Landscape (in the units of some real-world coordinate reference system, e.g. decimal degrees, or meters). This information is only used if GIS rasters of Landscape layers are to be written out as GIS raster files. Defaults to the meaningless value (0,0), and this value usually needn’t be changed in your parameters file, because it will be automatically updated to match the ULC value of any GIS rasters that are read in for use as Layers (assuming they all share the same ULC; otherwise, an Error is thrown).

```
prj

#projection of the Landscape
'prj': None,
```

str; (WKT projection string)
default: None
reset: N

This defines the projection of the Landscape, as a string of Well Known Text (WKT). This information is only used if GIS rasters of Landscape layers are to be written out as GIS raster files. Defaults to None, which is fine, because this value will be automatically updated to match the projection of any GIS rasters that are read in for us as Layers (assuming they all share the same projection; otherwise, an Error is thrown)

**Layers**

```
layers

"""""""""
### layers ###
"""""""""
'layers': {
    "layer name (LAYER NAMES MUST BE UNIQUE!)
    'layer_0': {

{str, int}
default: layer_<n>
reset? P

This parameter defines the name for each Layer. (Note that unlike most parameters, this parameter is a dict key, the value for which is a dict of parameters defining the Layer being named.) As the capitalized reminder in the parameters states, each Layer must have a unique name (so that a parameterized Layer isn’t overwritten in the ParametersDict by a second, identically-named Layer; Geonomics checks for unique names and throws an Error if this condition is not met. Layer names can, but needn’t be, descriptive of what each Layer represents. Example valid values include: 0, 0.1, ‘layer_0’, 1994, ‘1994’, ‘mean_ann_tmp’. Names default to layer_<n>, where n is a series of integers starting from 0 and counting the number of Layers.

**Init**

There are four different types of Layers that can be created. The parameters for each are explained in the next four subsections.
random

n_pts

```python
#parameters for a 'random'-type Layer
'rand': {
    #number of random points
    'n_pts': 500,
}
```

int
default: 500
reset? P

This defines the number of randomly located, randomly valued points from which the random Layer will be interpolated. (Locations drawn from uniform distributions between 0 and the Landscape dimensions on each axis. Values drawn from a uniform distribution between 0 and 1.)

interp_method

```python
#interpolation method ('linear', 'cubic', or 'nearest')
'interp_method': 'linear',
```

```
{'linear', 'cubic', 'nearest'}
default: 'linear'
reset? N
```

This defines the method to use to interpolate random points to the array that will serve as the Layer’s raster. Whichever of the three valid values is chosen (‘linear’, ‘cubic’, or ‘nearest’) will be passed on as an argument to scipy.interpolate.griddata. Note that the ‘nearest’ method will generate a random categorical array, such as might be used for modeling habitat types.

defined

rast

```python
#parameters for a 'defined'-type Layer
'defined': {
    #raster to use for the Layer
    'rast': np.ones((100,100)),
}
```

nx2 np.ndarray
default: np.ones((100,100))
reset? Y

This defines the raster that will be used for this Layer. Can be set to None if an array for the raster should instead be interpolated from a set of valued points using the pts, vals, and interp_method parameters. Dimensions of this array must match the dimensions of the Landscape.

pts
parameters for a 'defined'-type Layer
'defined': {
    # point coordinates
    'pts': None,
    # point values
    'vals': None,
    # interpolation method
    'interp_method': None,
}

This defines the coordinates of the points to use to interpolate this Layer. Can be left as None if the rast parameter is given a numpy.ndarray.

This defines the values of the points to use to interpolate this Layer. Can be left as None if the rast parameter is given a numpy.ndarray.

This defines the method to use to interpolate random points to the array that will serve as the Layer’s raster. Whichever of the valid string values is chosen (‘linear’, ‘cubic’, or ‘nearest’) will be passed on as an argument to scipy.interpolate.griddata. Note that the ‘nearest’ method will generate a random categorical array, such as might be used for modeling habitat types. Can be left as None if the rast parameter is given a numpy.ndarray.

parameters for a 'file'-type Layer
'file': {
    #</path/to/file>.<ext>
    'filepath': '/PATH/TO/FILE.EXT',
}

This defines the file path to use for this Layer. The file must have an associated extension that is included in the output array defined by the specify rast parameter.
This defines the location and name of the file that should be read in as the raster-array for this Layer. Valid file types include a `.txt` file containing a 2d np.ndarray, or any GIS raster file that can be read by rasterio.open. In all cases, the raster-array read in from the file must have dimensions equal to the stipulated dimensions of the Landscape (as defined in the dims parameter, above); otherwise, Geonomics will throw an Error. Defaults to a dummy filename that must be changed.

`scale_min_val`

```
#minimum value to use to rescale the Layer to [0,1]
'scale_min_val': None,
```

{float, int}

default: None

reset? P

This defines the minimum value (in the units of the variable represented by the file you are reading in) to use when rescaling the file’s array to values between 0 and 1. (This is done to satisfy the requirement that all Geonomics Layers have arrays in that interval). Defaults to None (in which case Geonomics will set it to the minimum value observed in this file’s array). But note that you should put good thought into this parameter, because it won’t necessarily be the minimum value observed in the file; for example, if this file is being used to create a Layer that will undergo environmental change in your Model, causing its real-world values to drop below this file’s minimum value, then you will probably want to set this value to the minimum real-world value that will occur for this Layer during your Model scenario, so that low values that later arise on this Layer don’t get truncated at 0.

`scale_max_val`

```
#maximum value to use to rescale the Layer to [0,1]
'scale_max_val': None,
```

{float, int}

default: None

reset? P

This defines the maximum value (in the units of the variable represented by the file you are reading in) to use when rescaling the file’s array to values between 0 and 1. (This is done to satisfy the requirement that all Geonomics Layers have arrays in that interval). Defaults to None (in which case Geonomics will set it to the maximum value observed in this file’s array). But note that you should put good thought into this parameter, because it won’t necessarily be the maximum value observed in the file; for example, if this file is being used to create a Layer that will undergo environmental change in your Model, causing its real-world values to increase above this file’s maximum value, then you will probably want to set this value to the maximum real-world value that will occur for this Layer during your Model scenario, so that high values that later arise on this Layer don’t get truncated at 1.

`coord_prec`

```
#decimal-precision to use for coord-units (ulc & res)
'coord_prec': 5,
```

int

default: 5

reset? P
This defines number of decimals to which to round upper-left corner coordinates and resolution values read in from a raster file. Because Geonomics requires equality of these values amongst all input raster files, this allows the user to stipulate the level of precision of their coordinate system, avoiding false coordinate-system mismatch errors because of arbitrary float imprecision. (Note that for Layers for which change rasters will be read in, the same coordinate precision value will be used for all input rasters.)

**units**

```yaml
#units of this file's variable
'units': None,
```

{str, None}
default: None
reset? P

This is an optional parameter providing a string-representation of the units in which a raster file’s variable is expressed. If provided, it will be used to label the colorbar on plots of the raster’s Layer.

**nlmepy**

**function**

```yaml
#parameters for an 'nlmepy'-type Layer
'nlmepy': {
    #nlmepy function to use to create this Layer
    'function': 'mpd',
}
```

str that is the name of an nlmepy function
default: 'mpd'
reset? P

This indicates the nlmepy function that should be used to generate this Layer’s array. (nlmepy is a Python package for generating neutral landscape models; NLMs.) Defaults to 'mpd' (the function for creating a midpoint-displacement NLM). Can be set to any other str that identifies a valid nlmepy function, but then the remaining parameters in this section must be changed to the parameters that that function needs, and only those parameters (because they will be unpacked into this function, i.e. passed on to it, at the time it is called. (Visit the Cheese Shop for more information about the nlmepy package and available functions).

**nRow**

```yaml
#number of rows (MUST EQUAL LAND DIMENSION y!)
'nRow': 20,
```

int
default: 20
reset? P

This defines the number of rows in the nlmepy array that is created. As the capitalized reminder in the parameters file mentions, this must be equal to the y-dimension of the Landscape; otherwise, an error will be thrown. Note that this parameter (as for the remaining parameters in this section, other than the function parameter) is valid for the default nlmepy.mpd function that is set by the function parameter; if you are using a different nlmepy function to create this
Layer then this and the remaining parameters must be changed to the parameters that that function needs, and only those parameters (because they will be unpacked into that function, i.e. passed on to it, at the time it is called).

**nCol**

```plaintext
#number of cols (MUST EQUAL LAND DIMENSION x!)
'nCol': 20,
```

**h**

```plaintext
#level of spatial autocorrelation in element values
'h': 1,
```

**Change**

**change_rast**

```plaintext
#land-change event for this Layer
'change': {
    #array of file for final raster of event, or directory
    #of files for each stepwise change in event
    'change_rast': '/PATH/TO/FILE.EXT',
}
```

This defines either the final raster of the Landscape change event (with valid values being a numpy.ndarray or a string pointing to a valid raster file, i.e. a file that can be read by rasterio.open); or the stepwise series of changes to be made over the course of the Landscape change event (with the valid value being a string pointing to a directory.
full of valid raster files). Note that whether an array, a raster, or multiple rasters are input, their dimensions must be

equal to the dimensions of the Layer that is being changed (and hence to the Landscape to which it belongs).

Also note that if a directory of stepwise-change rasters is provided, the rasters’ filenames must begin with the integer
timesteps at which they should be used during the change event, followed by underscores. (For example, files with the
filenames ‘50_mat_2001.tif’, ‘60_mat_2011.tif’, ‘65_mat_2011.tif’ would be used at timesteps 50, 60, and 65 during
a model.) Defaults to a dummy file name that must be changed.

**start_t**

```python
#starting timestep of event
'start_t': 50,
```

int
default: 50
reset? P

This indicates the first timestep of the Landscape-change event. Defaults to 50, but should be set to suit your specific
scenario. If a directory of files is provided for the change_rast parameter, then this must match the earliest timestep
in that series of files (as indicated by the integers at the beginning of the file names).

**end_t**

```python
#ending timestep of event
'end_t': 100,
```

int
default: 100
reset? P

This indicates the last timestep of the Landscape-change event. Defaults to 100, but should be set to suit your specific
scenario. If a directory of files is provided for the change_rast parameter, then this must match the final
timestep in that series of files (as indicated by the integers at the beginning of the file names).

**n_steps**

```python
#number of stepwise changes in event
'n_steps': 5,
```

int
default: 5
reset? P

This indicates the number of stepwise changes to use to model a Landscape-change event. If the the change_rast
parameter is a directory of files, then the value of this parameter must be the number of files in that directory. If the
change_rast parameter is either an np.ndarray or a file name, then the changes during the Landscape-change
event are linearly interpolated (cellwise for the whole Layer) to this number of discrete, instantaneous Landscape
changes between the starting and ending rasters. Thus, the fewer the number of steps, the larger, magnitudinally,
each change will be. So more steps may be ‘better’, as it will better approximate change that is continuous in time.
However, there is a potentially significant memory trade-off here: The whole series of stepwise-changed arrays is
computed when the Model is created, then saved and used at the appropriate timestep during each Model run (and
if the Layer that is changing is used by any Species as a _ConductanceSurface then each intermediate
_ConductanceSurface is also calculated when the Model is first built, which can be much more memory-
intensive because these are 3-dimensional arrays). These objects take up memory, which may be limiting for larger
Models and/or Landscape objects. This often will not be a major issue, but depending on your use case it could
pose a problem, so is worth considering.
3.9.2 Community and Species parameters

Species

spp_<n>

```python
#sp name (SPECIES NAMES MUST BE UNIQUE!)
'spp_0': {

{str,int}
default: spp_<n>
reset? P
```

This parameter defines the name for each Species. (Note that unlike most parameters, this parameter is a dict key, the value for which is a dict of parameters defining the Species being named.) As the capitalized reminder in the parameters states, each Species must have a unique name (so that a parameterized Species isn’t overwritten in the ParametersDict by a second, identically-named Species; Geonomics checks for unique names and throws an Error if this condition is not met. Species names can, but needn’t be, descriptive of what each Species represents. Example valid values include: 0, ‘spp0’, ‘high-dispersal’, ‘C. fasciata’. Names default to spp_<n>, where n is a series of integers starting from 0 and counting the number of Species.

Init

N

```python
'init': {
    #starting number of individs
    'N': 250,
```

int
default: 250
reset? P

This defines the starting size of this Species. Importantly, this may or may not be near the stationary size of the Species after the Model has burned in, because that size will depend on the carrying-capacity raster (set by the K parameter), and on the dynamics of specific a Model (because of the interaction of its various parameters).

K_layer

```python
#name of the carrying-capacity Layer
'K_layer': 'layer_0',
```

str
default: ‘layer_0’
reset? P

This indicates, by name, the Layer to be used as the carrying-capacity raster for a Species. The values of this Layer, multiplied by K_factor, should express the carrying capacity at each cell, in number of Individuals.
Note that the sum of the values of the product of this Layer and K_factor can serve as a rough estimate of the expected stationary number of individuals of a Species; however, observed stationary size could vary substantially depending on various other Model parameters (e.g. birth and death rates and mean number of offspring per mating event) as well as on stochastic events (e.g. failure to colonize, or survive in, all habitable portions of the Landscape).

**K_factor**

```
# multiplicative factor for carrying-capacity layer
'K_factor': 1,
```

{int, float}
default: 1
reset? P

This defines the factor by which the raster of the Layer indicated by K_layer will be multiplied to create a Species' carrying- capacity raster. Because Layers' rasters are constrained to [0,1], this allows the user to stipulate that cells have carrying capacities in excess of 1.

**Mating**

**repro_age**

```
'mating': {
  # age(s) at sexual maturity (if tuple, female first)
  'repro_age': 0,
}
```

{int, (int, int), None}
default: 0
reset? P

This defines the age at which Individuals in the Species can begin to reproduce. If the value provided is a 2-tuple of different numbers (and the Species uses separate sexes), then the first number will be used as females' reproductive age, the second as males'. If the value is 0, or None, Individuals are capable of reproduction from time of time.

**sex**

```
# whether to assign sexes
'sex': False,
```

bool
default: False
reset? P

This determines whether Individuals will be assigned separate sexes that are used to ensure only male-female mating events.

**sex_ratio**

```
# ratio of males to females
'sex_ratio': 1/1,
```


This defines the ratio of males to females (i.e. it will be converted to a probability that an offspring is a male, which is used as the probability of a Bernoulli draw of that offspring’s sex).

```
#intrinsic growth rate
'R': 0.5,
```

This defines a Species' intrinsic growth rate, which is used as the ‘R’ value in the spatialized logistic growth equation that regulates population density

\[
\frac{dN_{x,y}}{dt} = r N_{x,y}(1 - \frac{N_{x,y}}{K_{x,y}})
\]

```
#intrinsic birth rate (MUST BE 0<=b<=1)
'b': 0.2,
```

This defines a Species' intrinsic birth rate, which is implemented as the probability that an identified potential mating pair successfully produces offspring. Because this is a probability, as the capitalized reminder in the parameters file mentions, this value must be in the inclusive interval [0, 1].

NOTE: this may later need to be re-implemented to allow for spatial variation in intrinsic rate (i.e. expression of a birth-rate raster), and/or for density-dependent birth as well as mortality

```
#expectation of distr of n offspring per mating pair
'n_births_dist_lambda': 1,
```

This defines the lambda parameter for the Poisson distribution from which a mating pair’s number of offspring is drawn (unless n_births_fixed is set to True, in which case it defines the number of offspring produced by each successful mating event). Hence, this is either the expected or exact value for the number of offspring born in a successful mating event (depending on how n_births_fixed is set).

```
#whether n births should be fixed at n_births_dist_lambda
'n_births_fixed': True,
```
bool
default: True
reset? P

This determines whether or not the number of births for each mating event will be fixed. If set to true, each successful mating event will produce \texttt{n_births_distr\_lambda} new offspring.

\begin{verbatim}
#radius of mate-search area (None, for panmixia)
'mating_radius': 1

{float, int, None}
default: 1
reset? Y
\end{verbatim}

This defines the radius within which an \texttt{Individual} can find a mate. This radius is provided to queries run on the \_\texttt{KDTree} object. (If set to \texttt{None} then true panmixia will be used, i.e. each \texttt{Individual}, with probability equal to its \texttt{Species}' birth rate, will choose any other individual in the population as its mate, after which the chosen pair will then go through age- and sex-eligibility checks as needed given the parameterization.)

\begin{verbatim}
#whether individs should choose nearest neighs as mates
'choose\_nearest\_mate': False

py bool
default: False
reset? P
\end{verbatim}

This determines whether or each \texttt{Individual} will always choose its nearest neighbor as a mate. Defaults to False, allowing each focal \texttt{Individual} to randomly choose from among all other \texttt{Individuals} occurring within its mating radius. (In that case, if \texttt{inverse\_dist\_mating} is False then all other nearby \texttt{Individuals} will have equal probability of being chosen; if \texttt{inverse\_dist\_mating} is True, then other \texttt{Individuals} will have probabilities linearly related to their inverse distance from the focal \texttt{Individual}.) (Note that this parameter will only be used if \texttt{mating\_radius} is not \texttt{None}.)

\begin{verbatim}
#whether mate-choice should be inverse distance-weighted
'inverse\_dist\_mating': False

py bool
default: False
reset? P
\end{verbatim}

This determines whether or each focal \texttt{Individual} will use the inverse of the distance between itself and all other \texttt{Individuals} occurring within its mating radius to weight the mutually exclusive probabilities of choosing each of those other \texttt{Individuals} as a mate. If False, then each other \texttt{Individual} within the focal \texttt{Individual}'s mating radius has a uniform probability of being chosen as a mate. (Note that this parameter will only be used if \texttt{choose\_nearest\_mate} is False and \texttt{mating\_radius} is not \texttt{None}.)
Mortality

max_age

#maximum age
'max_age': 1,

{int, None}
default: 1
reset? P

This defines the maximum age an individual can achieve before being forcibly culled from the Species. Defaults to 1 (which will create a Wright-Fisher-like simulation, with discrete generations). Can be set to any other age, or can be set to None (in which case no maximum age is enforced).

d_min

#min P(death) (MUST BE 0<=d_min<=1)
'd_min': 0,

float in interval [0, 1]
default: 0
reset? N

This defines the minimum probability of death that an Individual can face each time its Bernoulli death-decision is drawn. Because this is a probability, as the capitalized reminder in the parameters file mentions, this value must be in the inclusive interval [0, 1].

d_max

#max P(death) (MUST BE 0<=d_max<=1)
'd_max': 1,

float in interval [0, 1]
default: 1
reset? N

This defines the minimum probability of death that an Individual can face each time its Bernoulli death-decision is drawn. Because this is a probability, as the capitalized reminder in the parameters file mentions, this value must be in the inclusive interval [0, 1].

density_grid_window_width

'mortality': {
    #width of window used to estimate local pop density
    'dens_grid_window_width': None,
}

{float, int, None}
default: None
reset? N

This defines the width of the window used by the _DensityGridStack to estimate a raster of local Species densities. The user should feel free to set different values for this parameter (which could be especially helpful when calling Model.plot_density to inspect the resulting surfaces calculated at different window widths, if trying to
heuristically choose a reasonable value to set for a particular simulation scenario). But be aware that choosing particularly small window widths (in our experience, windows smaller than ~1/20th of the larger Landscape dimension) will cause dramatic increases in the run-time of the density calculation (which runs twice per timestep). Defaults to None, which will internally be set to the integer nearest to 1/10th of the larger Landscape dimension; for many purposes this will work, but in some cases the user may wish to control this.

Movement

```python
move

#whether or not the species is mobile
'move': True,
```

**py bool**

default: True
reset? P

This determines whether the :py: Species being parameterized is mobile (i.e. whether its individuals should move). A Species without movement will still undergo dispersal of offspring, but after dispersing those offspring will remain fixed in location until death.

direction_distr_mu

```python
'movement': {
    #mode of distr of movement direction
    'direction_distr_mu': 1,
}
```

{int, float}

default: 1
reset? N

This is the $\mu$ parameter of the VonMises distribution (a circularized normal distribution) from which movement directions are chosen when movement is random and isotropic (rather than being determined by a _ConductanceSurface; if a _ConductanceSurface is being used this parameter is ignored). The $\kappa$ value that is fed into this same distribution (direction_distr_kappa) causes it to be very dispersed, such that the distribution is effectively a uniform distribution on the unit circle (i.e. all directions are effectively equally probable). For this reason, changing this parameter without changing the direction_distr_kappa value also, will make no change in the directions drawn for movement. If random, isotropic movement is what you aim to model then there is probably little reason to change these parameters.

direction_distr_kappa

```python
#concentration of distr of movement direction
'direction_distr_kappa': 0,
```

{int, float}

default: 0
reset? N

This is the $\kappa$ parameter of the VonMises distribution (a circularized normal distribution) from which movement directions are chosen when movement is random and isotropic (rather than being determined by a _ConductanceSurface; if a _ConductanceSurface is being used this parameter is ignored). The default
value of 0 will cause this distribution to be very dispersed, approximating a uniform distribution on the unit circle and rendering the $\mu$ value (direction_distr_mu) effectively meaningless. However, as this parameter’s value increases the resulting circular distributions will become more concentrated around $\mu$, making the value fed to direction_distr_mu influential. If random, isotropic movement is what you aim to model then there is probably little reason to change these parameters.

**movement_distance_distr_param1**

```yaml
#1st param of distr of movement distance
'movement_distance_distr_param1': 0.1,
```

{int, float}

default: 0.1
reset? Y

This is the first parameter of the distribution used to draw movement distances. The values generated by the movement distribution will be expressed in units of raster-cell widths. This parameter and movement_distance_distr_param2 should be set to reflect a distribution of movement distances that is appropriate for your scenario. The distribution to which this parameter applies depends on the value of the movement_distance_distr parameter.

**movement_distance_distr_param2**

```yaml
#2nd param of distr of movement distance
'movement_distance_distr_param2': 0.5,
```

{int, float}

default: 0.5
reset? Y

This is the second parameter of the distribution used to draw movement distances. The values generated by the movement distribution will be expressed in units of raster-cell widths. This parameter and movement_distance_distr_param1 should be set to reflect a distribution of movement distances that is appropriate for your scenario. The distribution to which this parameter applies depends on the value of the movement_distance_distr parameter.

**movement_distance_distr**

```yaml
#movement distance distr to use ('lognormal', 'levy', 'wald')
'movement_distance_distr': 'lognormal',
```

str

default: ‘lognormal’
reset? Y

This determines whether movement is modeled using a lognormal distribution (‘lognormal’; default), a Lévy distribution (‘levy’), or a Wald distribution (‘wald’).

**dispersal_distance_distr_param1**

```yaml
#1st param of distr of dispersal distance
'dispersal_distance_distr_param1': -1,
```

{int, float}

default: -1
reset? Y

3.9. Parameters
This is the first parameter of the distribution used to draw dispersal distances. The values generated by the dispersal distribution will be expressed in units of raster-cell widths. This parameter and `dispersal_distance_distr_param2` should be set to reflect a distribution of dispersal distances that is appropriate for your scenario. The distribution to which this parameter applies depends on the value of the `dispersal_distance_distr` parameter.

`dispersal_distance_distr_param2`

```plaintext
#2nd param of distr of dispersal distance
'dispersal_distance_distr_param2': 0.05,
```

{int, float}
default: 0.05
reset? Y

This is the second parameter of the distribution used to draw dispersal distances. The values generated by the dispersal distribution will be expressed in units of raster-cell widths. This parameter and `dispersal_distance_distr_param1` should be set to reflect a distribution of dispersal distances that is appropriate for your scenario. The distribution to which this parameter applies depends on the value of the `dispersal_distance_distr` parameter.

`dispersal_distance_distr`

```plaintext
#dispersal distance distr to use ('lognormal', 'levy', 'wald')
'dispersal_distance_distr': 'lognormal',
```

str
default: ‘lognormal’
reset? Y

This determines whether dispersal is modeled using a lognormal distribution (‘lognormal’; default), a Lévy distribution (‘levy’), or a Wald distribution (‘wald’).

---

**Movement and Dispersal _ConductanceSurfaces**

`layer`

```plaintext
'move_surf' : {
    #move-surf Layer name
    'layer': 'layer_0',
}
```

str
default: ‘layer_0’
reset? P

This indicates, by name, the Layer to be used as to construct the _ConductanceSurface for a Species. Note that this can also be thought of as the Layer that should serve as a Species’ permeability raster (because Individuals moving on this _ConductanceSurface toward the higher (if mixture distributions are used) or highest (if unimodal distributions are used) values in their neighborhoods).

`mixture`

```plaintext
#whether to use mixture distrs
'mixture': True,
```

---

40 Chapter 3. About the Documentation
bool
default: True
reset? P

This indicates whether the _ConductanceSurface should be built using VonMises mixture distributions or unimodal VonMises distributions. If True, each cell in the _ConductanceSurface will have an approximate circular distribution that is a weighted sum of 8 unimodal VonMises distributions (one per cell in the 8-cell neighborhood); each of those summed unimodal distributions will have as its mode the direction of the neighboring cell on which it is based and as its weight the relative permeability of the cell on which it is based (relative to the full neighborhood). If False, each cell in the _ConductanceSurface will have an approximated circular distribution that is a single VonMises distribution with its mode being the direction of the maximum-valued cell in the 8-cell neighborhood and its concentration determined by vm_distr_kappa.

vm_distr_kappa

```python
# concentration of distrs
'vm_distr_kappa': 12,
```

{int, float}
default: 12
reset? N

This sets the concentration of the VonMises distributions used to build the approximated circular distributions in the _ConductanceSurface. The default value was chosen heuristically as one that provides a reasonable concentration in the direction of a unimodal VonMises distribution’s mode without causing VonMises mixture distributions built from an evenly weighted sum of distributions pointing toward the 8-cell-neighborhood directions to have 8 pronounced modes. There will probably be little need to change the default value, but if interested then the user could create Models with various values of this parameter and then use the Model.plot_movement_surface method to explore the influence of the parameter on the resulting _ConductanceSurfaces.

approx_len

```python
# length of approximation vectors for distrs
'approx_len': 5000,
```

{int}
default: 5000
reset? P

This determines the length of the vector of values used to approximate each distribution on the _ConductanceSurface (i.e. the size of the z-axis of the np.ndarray used to hold all the distribution-approximations, where the y and x axes have the same dimensions as the Landscape). The default value of 5000 is fine for many cases, but may need to be reduced depending on the Landscape dimensions (because for a larger Landscape, say 1000x1000 cells, it would create a _ConductanceSurface that is roughly 4Gb, and if the Layer on which the _ConductanceSurface is based will be undergoing landscape changes then numerous versions of an object of this size would need to be generated when the Model is built and held in memory). The value to use for this parameter will depend on the size of the Landscape, the exact scenario being simulated, and the memory of the machine on which the Model is to be run.

_genomicArchitecture

gen_arch_file
The arguments indicated whether a custom genomic architecture file should be used to create a Species' GenomicArchitecture, and if so, where that file is located. If the value is None, no file will be used and the values of this Species' other genomic architecture parameters in the parameters file will be used to create the GenomicArchitecture. If the value is a str pointing to a custom genomic-architecture file (i.e. a CSV file with loci as rows and 'locus_num', 'p', ‘dom’, ‘r’, ‘trait’, and ‘alpha’ as columns stipulating the starting allele frequencies, dominance values, inter-locus recombination rates, trait names, and effect sizes of all loci; values can be left blank if not applicable). Geonomics will create an empty file of this format for each Species for which the 'genomes' argument is given the value 'custom' when gnx.make_parameters_file is called (which will be saved as '<your_model_name>_spp-<n>_gen_arch.csv').

Note that when Geonomics reads in a custom genomic architecture file to create a Model, it will check that the length (i.e. number of rows) in this file is equal to the length stipulated by the L parameter, and will also check that the first value at the top of the 'r' column is 0.5 (which is used to implement independent assortment during gametogenesis). If either of these checks fails, Geonomics throws an Error.

L

#num of loci
'L': 1000,

int
default: 1000
reset? P

This defines the total number of loci in the genomes in a Species.

l_c

#num of chromosomes
'l_c': [100],

list of ints
default: [100]
reset? P

This defines the lengths (in number of loci) of each of the chromosomes in the genomes in a Species. Note that the sum of this list must equal L, otherwise Geonomics will throw an Error. Also note that Geonomics models genomes as single L x 2 arrays, where separate chromosomes are delineated by points along the genome where the recombination rate is 0.5; thus, for a model where recombination rates are often at or near 0.5, this parameter will have little meaning.

start_p_fixed

#fixed starting allele freq; None/False → rand; True → 0.5
'start_p_fixed': 0.5,
If a float on the interval [0,1] is provided, that value will be used as the starting allele frequency at which all loci (except neutral loci, if start_neut_zero is True) will be fixed. If None, the starting allele frequency of each locus will be drawn as a uniform random variable between 0 and 1, inclusive. If a bool is provided, True will fix all loci at the default starting allele frequency of 0.5, whereas False will not fix starting allele frequencies, effectively having the same effect as None. Defaults to 0.5.

**start_neut_zero**

```python
#whether to start neutral locus freqs at 0
'start_neut_zero': False,
```

bool

default: False

reset? P

If True, all neutral loci will start with the ‘1’ allele at a frequency of 0 (i.e. all individuals will be homozygous ‘0’|’0’ at those loci).

**mu_neut**

```python
#genome-wide per-base neutral mut rate (0 to disable)
'mu_neut': 0,
```

float

default: 1e-9

reset? P

This defines the genome-wide per-base neutral mutation rate. This value can be set to 0 to disable neutral mutation.

**mu_delet**

```python
#genome-wide per-base deleterious mut rate (0 to disable)
'mu_delet': 0,
```

float

default: 0

reset? P

This defines the genome-wide per-base deleterious mutation rate. This value can be set to 0 to disable deleterious mutation. Note that all deleterious mutation will fall outside the loci that affect any Trait a Species may have, and will behave simply as globally deleterious mutations (i.e. mutations that reduce the mutated Individual’s fitness regardless of that Individual’s spatial location).

**delet_alpha_distr_shape**

```python
#shape of distr of deleterious effect sizes
deleat_delet_alpha_distr_shape': 0.2,
```

float

default: 0.2
reset? P
This defines the shape parameter of the gamma distribution from which the effect sizes of deleterious loci are drawn. (Values drawn will be truncated to the interval [0,1].)

**delet_alpha_distr_scale**

```
#scale of distr of deleterious effect sizes
'delet_alpha_distr_scale': 0.2,
```

float
default: 0.2
reset? P
This defines the scale parameter of the gamma distribution from which the effect sizes of deleterious loci are drawn. (Values drawn will be truncated to the interval [0,1].)

**r_distr_alpha**

```
#alpha of distr of recomb rates
'r_distr_alpha': None,
```

{float, None}
default: 0.5
reset? P
This defines the alpha parameter of the beta distribution from which interlocus recombination rates are drawn. (Values drawn will be truncated to the interval [0, 0.5].) If **r_distr_beta** is None, recombination rates will be fixed at this value. (Defaults to 0.5, and **r_distr_beta** defaults to None, such that all loci will be independent by default.) If set to None, all recombination rates will be fixed at a value (1/L) that yields approximately 1 expected recombination event per gamete per generation.

**r_distr_beta**

```
#beta of distr of recomb rates
'r_distr_beta': None,
```

{float, None}
default: None,
reset? P
This defines the beta parameter of the beta distribution from which interlocus recombination rates are drawn. (Values drawn will be truncated to the interval [0, 0.5].) Defaults to None, which will fix recombination rates at the value of **r_distr_alpha** (which defaults to 0.5, i.e. independence), or else will fix all rates at a value (1/L) that yields approximately 1 expected recombination event per gamete per generation.

**dom**

```
#whether loci should be dominant (for allele '1')
'dom': False,
```

bool
default: False
reset? P
This indicates whether loci should be treated as dominant (if True) for the ‘1’ allele or as codominant (if False). Codominance is the default behavior, because it is assumed that Geonomics will often be used to model quantitative traits, for which this is a reasonable assumption.

**pleiotropy**

```python
#whether to allow pleiotropy
'pleiotropy': False,
```

**bool**

default: False

reset? P

This indicates whether pleiotropy should be allowed. If True, loci will be permitted to contribute to more than one Trait.

**recomb_rate_custom_fn**

```python
#custom fn for drawing recomb rates
'recomb_rate_custom_fn': None,
```

**{function,None}**

default: None

reset? P

This parameter allows the user to provide a custom function according to which interlocus recombination rates will be assigned. If set to None, the default behavior (i.e. recombination rates chosen from a beta distribution using `r_distr_alpha` and `r_distr_beta`) will be used.

**n_recomb_paths_mem**

```python
#number of recomb paths to hold in memory
'n_recomb_paths_mem': int(1e4),
```

**int**

default: int(1e4)

reset? P

This defines the maximum number of recombination paths for Genomics to hold in memory at one time. Geonomics models recombination by using the interlocus recombination rates to draw a large number of recombination ‘paths’ along the Lx2 genome array (when the `Model` is first built), and then shuffling and cycling through those recombination paths as needed during `Model` runs. Of the total number of paths created, some subset will be held in memory (the number of these is defined by this parameter), while the remainder will live in a temporary file (which is occasionally read in whenever the paths in memory are close to being used up). Thus, to avoid problems, the number provided to this parameter should be comfortably larger than the largest anticipated number of recombination paths that will be needed during a single mating event (i.e. larger than two times the largest anticipated number of offspring to be born to the `Species` during one timestep).

**n_recomb_paths_tot**

```python
#total number of recomb paths to simulate
'n_recomb_paths': int(1e5),
```

**int**

default: int(1e5)

reset? P

This defines the total number of recombination paths that Geonomics will generate. Geonomics models recombination by using the interlocus recombination rates to draw a large number of recombination ‘paths’ along the Lx2 genome array (when the `Model` is first built), and then shuffling and cycling through those recombination paths as needed.
during Model runs. The larger the total number of these paths that is created, the more closely Geonomics will model truly free recombination and the more precisely it will model the exact interlocus recombination rates defined in a Species' GenomicArchitecture.

**allow_ad_hoc_recomb**

```yaml
# whether to generate recombination paths at each timestep
'allow_ad_hoc_recomb': False,
```

bool
default: False
reset? P

This determines whether or not recombinants should be drawn each timestep (rather than recombination paths being drawn and stored when a model is first built, then used randomly through the model run). This is advantageous because it models recombination exactly (rather than approximating recombination by drawing some number of fixed recombination paths that get repeatedly used), and for combinations of larger genome sizes (L) and larger mean population sizes (N) it avoids the memory used by storing so many recombination paths drawn at model creation, thus making these parameterizations feasible on computers with memory limitations. It is disadvantageous, however, because it runs somewhat slower than the default approach (recombinants drawn at model creation) for a range of L and N values, and also because it is only available for parameterizations with homogeneous recombination across the genome.

**jitter_breakpoints**

```yaml
# whether to jitter recomb bps, to correctly track num_trees
'jitter_breakpoints': False,
```

bool
default: False
reset? P

This determines whether or not the recombination breakpoints stored by the tskit TableCollection should be slightly jitted off of their x.5 default positions. Enabling this will render each recombination effectively unique, and thus will allow the tskit.TreeSequence to correctly report the number of trees (TreeSequence.num_trees). However, it will do this at the expense of additional memory usage, which could potentially be limiting. Thus, this should be set to True if and only if the TreeSequence information will be used in a way that requires accurate representation of the number of trees.

**mut_log**

```yaml
# whether to save mutation logs
'mut_log': None,
```

{str,None}
default: None
reset? P

This indicates the location of the mutation-log file where Geonomics should save a record of each mutation that occurs for a Species Species, for each iteration. If None, no mutation log will be created and written to.
Traits

trait_<n>

#trait name (TRAIT NAMES MUST BE UNIQUE!)
'trait_0':

{str, int}
default: trait_<n>
reset? P

This parameter defines the name for each Trait. (Note that unlike most parameters, this parameter is a dict key, the value for which is a dict of parameters defining the Trait being named.) As the capitalized reminder in the parameters states, each Trait must have a unique name (so that a parameterized Trait isn’t overwritten in the ParametersDict by a second, identically-named Trait; Geonomics checks for unique names and throws an Error if this condition is not met. Trait names can, but needn’t be, descriptive of what each Trait represents. Example valid values include: 0, ‘trait0’, ‘tmp_trait’, ‘bill length’. Names default to trait_<n>, where n is a series of integers starting from 0 and counting the number of Traits for this Species.

layer

#trait-selection Layer name
'layer':

str
default: ‘layer_0’
reset? P

This indicates, by name, the Layer that serves as the selective force acting on this Trait. (For example, if this Trait is selected upon by annual mean temperature, then the name of the Layer representing annual mean temperature should be provided here.)

phi

#phenotypic selection coefficient
'phi':

{float, np.ndarray of floats}
default: 0.05
reset? P

This defines the phenotypic selection coefficient on this Trait (i.e the selection coefficient acting on the phenotypes, rather than the genotypes, of this Trait). The effect of this value can be thought of as the reduction (from 1) in an Individual’s survival probability when that Individual is maximally unfit (i.e. when that Individual has a phenotypic value of 1.0 but is located in a location with an environmental value of 0.0, or vice versa). When the value is a float then the strength of selection will be the same for all locations on the Landscape. When the value is an np.ndarray of equal dimensions to the Landscape then the strength of selection will vary across space, as indicated by the values in this array (what Geonomics refers to as a “spatially contingent” selection regime). Importantly, most individuals will experience selection on a given trait that is only a fraction of the strength dictated by this parameter, because an individual’s fitness for a given trait is determined by the product of the trait’s selection coefficient and the individual’s degree of mismatch to its local environment, and it is often unlikely that individuals would occur in environments that are completely opposed to those individuals’ phenotypic values (e.g. a 0-valued individual in a 1-valued environmental cell). Because of this, selection coefficients that would be considered ‘strong’ in classical,
aspatial population genetics models will tend to behave less strongly in Geonomics models. (For mathematical detail, see the Selection section.)

**n_loci**

```plaintext
#number of loci underlying trait
'n_loci': 1,
```

**mu**

```plaintext
#mutation rate at loci underlying trait
'mu': 1e-9,
```

This defines the number of loci that should contribute to the phenotypes of this Trait. These loci will be randomly drawn from across the genome.

This defines the mutation rate for this Trait (i.e. the rate at which mutations that affect the phenotypes of this Trait will arise). Set to 0 to disable mutation for this Trait.

**alpha_distr_mu**

```plaintext
#mean of distr of effect sizes
'alpha_distr_mu': 0.1,
```

This defines the mean of the normal distribution from which a Trait's initially parameterized loci and new mutations' effect sizes are drawn (with the exception of monogenic traits, whose starting locus always has an alpha value of 0.5, but whose later mutations are influenced by this parameter). For effect sizes drawn from a distribution, it is recommended to set this value set to 0 and adjust **alpha_distr_sigma**. For fixed effect sizes, set this value to the fixed effect size and set **alpha_distr_sigma** to 0; effects will alternate between positive and negative when they are assigned to loci. In either case, new mutations in a Trait will then be equally likely to decrease or increase Individuals' phenotypes from the multigenic baseline phenotype of 0.5 (which is also the central value on a Geonomics Landscape). It is also recommended that the user consider the number of loci for a trait when setting the fixed or distributed effect sizes; for example, for a trait with 10 underlying loci, an average or fixed absolute effect size of 0.1 will enable phenotypes that cover the range of values on a Geonomics Landscape (i.e. phenotypes 0 <= z <= 1), whereas 0.05 will likely not enable that full range of phenotypes, and 0.5 will generate many phenotypes that fall outside that range and will be selected against at all locations on the Landscape.

**alpha_distr_sigma**

```plaintext
#variance of distr of effect size
'alpha_distr_sigma': 0,
```

This defines the distribution of effect sizes for a Trait on a Geonomics Landscape.
This defines the standard deviation of the normal distribution from which a Trait’s new mutations’ effect sizes are drawn. For effect sizes drawn from a distribution, it is recommended to set this value set to some nonzero number and set alpha_distr_mu to 0. For fixed effect sizes, set this value to 0 and set alpha_distr_mu to the fixed effect size; effects will alternate between positive and negative when they are assigned to loci. In either case, new mutations in a Trait will then be equally likely to decrease or increase Individuals’ phenotypes from the multigenic baseline phenotype of 0.5 (which is also the central value on a Geonomics Landscape).

**max_alpha_mag**

```
#max allowed magnitude for an alpha value
'max_alpha': None,
```

default: None
reset? P

This defines the maximum value that can be drawn for a locus’ effect size (i.e. alpha). Defaults to None, but the user may want to set this to some reasonable value, to prevent chance creation of loci with extreme effects.

**gamma**

```
#curvature of fitness function
'gamma': 1,
```

default: 1
reset? N

This defines the curvature of the fitness function (i.e. how fitness decreases as the absolute difference between an Individual’s optimal and actual phenotypes increases). The user will probably have no need to change this from the default value of 1 (which causes fitness to decrease linearly around the optimal phenotypic value). Values < 1 will cause the fitness function to be concave up; values > 1 will cause it to be concave down.

**univ_adv**

```
#whether the trait is universally advantageous
'univ_adv': False
```

bool
default: False
reset? P

This indicates whether whether selection on a Trait should be universal (i.e. whether a phenotype of 1 should be optimal everywhere on the Landscape). When set to True, selection of the Trait will be directional on the entire Species, regardless of Individuals’ spatial contexts.
Species change

Demographic change

kind

```
#kind of event ('monotonic', 'stochastic',
#'cyclical', 'custom')
'kind': 'monotonic',
```

{'monotonic', 'stochastic', 'cyclical', 'custom'}

default: 'monotonic'

reset? P

This indicates what type of demographic change is being parameterized. Each event has a certain length (in timesteps; defined by the start and end parameters). Note that of the other parameters in this section, only those that are necessary to parameterize the type of change event indicated here will be used.

In ‘monotonic’ change events, a Species’ carrying capacity raster (K) is multiplied by a constant factor (rate) at each timestep during the event. In ‘stochastic’ change events, K fluctuates around the baseline value (i.e. the K-raster at the time that the change event begins) at each required timestep during the event (where the sizes of the fluctuations are drawn from the distribution indicated by distr, the floor and ceiling on those sizes are set by size_range, and the required timesteps are determined by interval). In ‘cyclical’ change events, K undergoes a number (indicated by n_cycles) of sinusoidal cycles between some minimum and maximum values (indicated by size_range). In ‘custom’ change events, the baseline K is multiplied by a series of particular factors (defined by sizes) at a series of particular timesteps (defined by timesteps).

start_t

```
#starting timestep
'start_t': 50,
```

int
default: 50
reset? P

This indicates the timestep at which the demographic change event should start.

end_t

```
#ending timestep
'end_t': 100,
```

int
default: 100
reset? P

This indicates the last timestep of the change event.

rate

```
#rate, for monotonic change
'rate': 1.02,
```

float
default: 1.02
reset? P
This indicates the rate at which a 'monotonic' change event should occur. At each timestep during the event, a new carrying capacity raster (K) will be calculated by multiplying the previous step’s K by this factor. Thus, values should be expressed relative to 1.0 indicating no change.

**interval**

```
#interval of changes, for stochastic change
'interval': 1,
```

int
default: 1
reset? P
This indicates the interval at which fluctuations should occur during a ‘stochastic’ change event (i.e. the number of timesteps to wait between fluctuations).

**distr**

```
#distr, for stochastic change {'uniform', 'normal'}
'distr': 'uniform',
```

{‘uniform’, ‘normal’}
default: ‘uniform’
reset? P
This indicates the distribution from which to draw the sizes of fluctuations in a ‘stochastic’ change event. Valid options are ‘uniform’ and ‘normal’.

**n_cycles**

```
#num cycles, for cyclical change
'n_cycles': 10,
```

int
default: 10
reset? P
This indicates the number of cyclical fluctuations that should occur during a ‘cyclical’ change event.

**size_range**

```
#min & max sizes, for stochastic & cyclical change
'size_range': (0.5, 1.5),
```

tuple of floats
default: (0.5, 1.5)
reset? P
This defines the minimum and maximum sizes of fluctuations that can occur during ‘stochastic’ and ‘cyclical’ change events.

**timesteps**

3.9. Parameters
list of ints  
default: [50, 90, 95]  
reset? P  
This defines the series of particular timesteps at which fluctuations should occur during a ‘custom’ change event.

sizes  
list of floats  
default: [2, 5, 0.5]  
reset? P  
This defines the series of particular fluctuations that should occur during a ‘custom’ change event.

Life-history change

$\langle\text{life_hist_param}\rangle$

str  
default: ‘$\langle\text{life_hist_param}\rangle$’  
reset? P  
This indicates the life-history parameter to be changed by this life-history change event. (Note that unlike most parameters, this parameter is a dict key, the value for which is a dict of parameters controlling how the life-history parameter that is named will change.)

timesteps

list of ints  
default: []  
reset? P  
This indicates the timesteps at which the life-history parameter being changed should change (to the values indicated by vals).

vals

list of floats  
default: []  
reset? P  
This indicates the values at which the life-history parameter being changed should change (to the values indicated by timesteps).
list of floats
default: []
reset? P
This indicates the values to which the life-history parameter being changed should change (at the timesteps indicated by timesteps).

3.9.3 Other parameters

Main

T

```
#total Model runtime (in timesteps)
'T': 100,
```

int
default: 100
reset? Y
This indicates the total number of timesteps for which the main portion of a Model (i.e. the portion after the burn-in has completed) will be run during each iteration.

burn_T

```
#min burn-in runtime (in timesteps)
'burn_T': 30,
```

int
default: 30
reset? P
This indicates the minimum number of timesteps for which a Model's burn-in will run. (Note this is only a minimum because the test for burn-in completion includes a check that at least this many timesteps have elapsed, but also includes two statistical checks of stationarity of the size of each Species in a Community.)

num

```
#seed number
'num': None,
```

{int,None}
default: None
reset? P
This indicates whether or not to set the seeds of the random number generators (by calling np.random.seed and random.seed) before building and running a Model. If value is an integer, the seeds will be set to that value. If value is None, seeds will not be set.

tskit_simp_interval

3.9. Parameters
# time step interval for simplification of tskit tables
'tskit_simp_interval': 100,

int
default: 100
reset? N

This sets the interval, in timesteps, between subsequent tskit simplifications. Defaults to simplifying every 100 timesteps, as suggested by the tskit package authors (see here). This most likely need not be changed, but for simulations with especially large population and/or genome sizes the user may wish to experiment with reducing this interval so as to improve performance.

## Iterations

### num_iterations

`num_iterations`

'\texttt{n\_its}': 2,

int
default: 2
reset? Y

This indicates the number of iterations for which the \texttt{Model} should be run. (Note that for each iteration a separate subdirectory of data and stats will be written, if your \texttt{Model} has parameterized data and stats to be collected.)

### rand_landscape

`\texttt{rand\_landscape}`

'\texttt{rand\_landscape}': False,

bool
default: False
reset? P

This indicates whether the \texttt{Landscape} should be randomized for each iteration. If True, a new \texttt{Landscape} will be generated at the start of each iteration. If False, the \texttt{Landscape} from iteration 0 will be saved and reused for each subsequent iteration.

### rand_community

`\texttt{rand\_comm}`

'\texttt{rand\_comm}': False,

bool
default: False
reset? P

This indicates whether the \texttt{Community} should be randomized for each iteration. If True, a new \texttt{Community} will be generated at the start of each iteration. If False, the \texttt{Community} from iteration 0 will be saved and reused for each
subsequent iteration (and whether that Community is saved before or after being burned in will depend on the value provided to the repeat_burn parameter).

**repeat_burn**

```yaml
#whether to burn in each iteration
'repeat_burn': False,
```

bool
default: False
reset? P

This indicates whether a reused Community should be burned in separately for each iteration for which it is reused. If True, the Community from iteration 0 will be saved as soon as its instantiated, but will have a new burn-in run for each iteration in which it is used. If False, the Community from iteration 0 will be saved after its burn-in is complete, and then will only have the main portion of its Model run separately during each iteration. (Note that if rand_community is set to True then the value of this parameter will not be used.)

---

**Data**

**Sampling**

**scheme**

```yaml
#sampling scheme {'all', 'random', 'point', 'transect'}
'scheme': 'random',
```

{'all', 'random', 'point', 'transect'}
default: 'random'
reset? P

This indicates the sampling scheme to use when collecting data from a Model. Currently valid values include ‘all’, ‘random’, ‘point’, and ‘transect’.

With ‘all’, data will be collected for all Individuals at each sampling timestep. With ‘random’, data will be collected from a random sample of Individuals (of size indicated by parameter n) from anywhere on the Landscape. With ‘point’, data will be collected from random samples of size n within a certain distance (radius) of each of a set of particular points (points). With ‘transect’, a linear transect of some number of points (n_transect_points) between some endpoints (transect_endpoints) will be created, and then data will be collected from random samples of size n with a certain distance (radius) of each point along the transect.

**n**

```yaml
#sample size at each point, for point & transect sampling
'n': 250,
```

int
default: 250
reset? P

This indicates the total number of Individuals to sample each time data is collected (if scheme is ‘random’), or the number of Individuals to sample around each one of a set of points (if scheme is ‘point’ or ‘transect’).

---

### 3.9. Parameters
This parameter will only be used if `scheme` is ‘random’, ‘point’, or ‘transect’; otherwise it may be set to None.

### points

```
#coords of collection points, for point sampling
'points': None,
```

{tuple of 2-tuples, None}
default: None
reset? P

This indicates the points around which to sample Individuals for data collection. This parameter will only be used if `scheme` is ‘point’; otherwise it may be set to None.

### transect_endpoints

```
#coords of transect endpoints, for transect sampling
'transect_endpoints': None,
```

{(2-tuple of 2-tuples, None)}
default: None
reset? P

This indicates the endpoints between which to create a transect, along which Individuals will be sampled for data collection. This parameter will only be used if `scheme` is ‘transect’; otherwise it may be set to None.

### n_transect_points

```
#num points along transect, for transect sampling
'n_transect_points': None,
```

{int, None}
default: None
reset? P

This indicates the number of points to create on the transect along which Individuals will be sampled for data collection. This parameter will only be used if `scheme` is ‘transect’; otherwise it may be set to None.

### radius

```
#collection radius around points, for point & transect sampling
'radius': None,
```

{float, int, None}
default: None
reset? P

This indicates the radius around sampling points within which Individuals may be sampled for data collection. This parameter will only be used if `scheme` is ‘point’ or ‘transect’; otherwise it may be set to None.

### when

```
#when to collect data
'when': None,
```

This indicates the points around which to sample Individuals for data collection. This parameter will only be used if `scheme` is ‘point’; otherwise it may be set to None.
This indicates the timesteps during main Model iterations at which data should be collected (in addition to after the final timestep of each iteration, when data is always collected for any Model for which data collection is parameterized). If value is a non-zero int, it will be treated as a frequency at which data should be collected (e.g. a value of 5 will cause data to be collected every 5 timesteps). If value is a list of ints, they will be treated as the particular timesteps at which data should be collected. If value is 0 or None, data will be collected only after the final timestep.

include_landscape

bool
default: False
reset? P
This indicates whether to include the Landscape Layers among the data that is collected. If True, each Layer will be written to a raster or array file (according to the format indicated by geo_rast_format) each time data is collected.

include_fixed_sites

bool
default: False
reset? P
This indicates whether fixed sites (i.e. loci which are fixed for either the 0 or 1 allele) should be included in any VCF files that are written. Thus, this parameter is only relevant if ‘vcf’ is one of the genetic data formats indicated by gen_format.

Format

gen_format

#format for genetic data {'vcf', 'fasta'}
'gen_format': ['vcf', 'fasta'],

[‘vcf’, ‘fasta’, [‘vcf’, ‘fasta’]}
default: [‘vcf’, ‘fasta’]
reset? P
This indicates the format or formats to use for writing genetic data. Currently valid formats include ‘vcf’ and ‘fasta’ formats. Either or both formats may be specified; all formats that are specified will be written each time data is collected.

documentation version: 1.1.4

57
#format for vector geodata {'csv', 'shapefile', 'geojson'}
'geo_vect_format': 'csv',

{'csv', 'shapefile', 'geojson'}
default: 'csv'
reset? P
This indicates the format to use for writing geographic vector data (i.e. Individuals’ point locations). Currently valid formats include ‘csv’, ‘shapefile’, and ‘geojson’. Any one format may be specified.

geo_rast_format

#format for raster geodata {'geotiff', 'txt'}
'geo_rast_format': 'geotiff',

{'geotiff', 'txt'}
default: 'geotiff'
reset? P
This indicates the format to use for writing geographic raster data (i.e. Layer arrays). Currently valid formats include ‘geotiff’ and ‘txt’. Either format may be specified. Note that this parameter will only be used if the include_landscape parameter is set to True.

Stats

The stats parameters section has subsection for each statistic that Geonomics can calculate. (Currently valid statistics include: - ‘Nt’: number of individuals at timestep t - ‘het’: heterozygosity - ‘maf’: minor allele frequency - ‘mean_fit’: mean fitness of a Species - ‘ld’: linkage disequilibrium

There are only a few parameters, which are shared across all of those subsections, and each parameter always means the same thing despite which statistic it is parameterizing. Thus, hereafter we provide a single of each of those parameters are how it works, regardless of the statistic for which it used:

calc

#whether to calculate
'calc': True,

bool
default: (varies by statistic)
reset? P
This indicates whether or not a given statistic should be calculated. Thus, only those statistics whose calc parameters are set to True will be calculated and saved when their Model is run.

calculation frequency (in timesteps)

'freq': 5,
int
default: (varies by statistic)
reset? P
This indicates the frequency with which a given statistic should be calculated during each iteration (in timesteps). If set to 0, Geonomics will calculate and save this statistic for only the first and last timesteps of each iteration.

mean

```
#whether to mean across sampled individs
'mean': False,
```

bool
default: (varies by statistic, and only valid for certain statistics)
reset? P
For some statistics that produce a vector of values each timestep when they are collected (containing one value per Individual), such as heterozygosity, this indicates whether those values should instead be meaned and saved as a single value for each timestep.

3.10 API

3.10.1 geonomics package

Subpackages
geonomics.help package

Module contents

Geonomics Help (geonomics.help)

Memory Help

get_obj_size Get the full size of a complex object

Parameters Help

plot_movement Plot example movement tracks generated by the input movement parameters

Background information

Geonomics models can occupy a lot of memory, and also contain a large number of parameters to be customized. The functions in this module aim to provide some assistance in assessing Geonomics’ behavior, so as to help users better plan, introspect, and debug their models.
geonomics Documentation, Release 1.1.4

geonomics.ops package

Submodules

gеonomics.ops.change module

Classes and functions to implement landscape, demographic, and life-history parameter change operations.

geonomics.ops.demography module

Functions to implement demographic operations (birth and death).

geonomics.ops.mating module

Functions to implement mating operations.

geonomics.ops.movement module

Functions to implement movement and dispersal.

geonomics.ops.mutation module

Functions to implement mutation operations

geonomics.ops.selection module

Functions to implement natural selection operations (aside from actual mortality)

Module contents

gеonomics.sim package

Submodules

gеonomics.sim.burnin module

Functions for testing burn-in stationarity.

class geonomics.sim.burnin.SpatialTester(spp)
    Bases: object
    plot_diffs (spp, bins=50, alpha=0.5)
    run_test (num_timesteps_back, alpha=0.05)
    update (spp)
    update_counts_cell (i, j, v)
geonomics Documentation, Release 1.1.4

geonomics.sim.data module

Classes and functions for implementing data collection and output

geonomics.sim.model module

Defines the core Model class, as well as its public and private methods

class geonomics.sim.model.Model (name, params, verbose=False)

Bases: object

Hierarchical data structure containing a Landscape and a Community as attributes, and serving as an interface for all of the core simulation functionality (as methods to run simulations and visualize results). The principal Geonomics data structure.

Parameters

- name (str) – A character string to be assigned as the Model’s name.
- params (ParametersDict) – A ParametersDict object containing all of the parameter values needed instantiate the Model. (See the online documentation for a detailed discussion of all parameters.)
- verbose (bool, optional) – Whether or not to display information about the Model’s status during simulations. Defaults to False.

calc_stats()

get_cells (spp=0, individs=None)

Returns the cell coordinates of any or all Individuals in a Species

Parameters

- spp ((int, str), optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).
- individs (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the Individuals whose cell coordinates should be returned. If None, all Individuals’ cell coordinates will be returned.

Returns

Returns an nx2 numpy array, where n is the number of Individuals whose coordinates were requested, and the 2 columns correspond to the i then j (i.e. lat then lon) array coordinates.

Note: Columns are in the reverse order of the columns of the array returned by Model.get_coords(), to facilitate array-subsetting of the Landscape Layer raster arrays.

Note: Rows are sorted by index number of the Individuals, regardless of input index order.

Return type np.array

get_coords (spp=0, individs=None)

Returns the x,y coordinates of any or all Individuals in a Species

Parameters

- spp ((int, str), optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).
• **individs** *(iterable collection of ints, optional, default: None)* – If provided, indicates the integer indices of the Individuals whose coordinates should be returned. If None, all Individuals’ coordinates will be returned.

**Returns**

- **np.array** – Returns an nx2 numpy array, where n is the number of Individuals whose coordinates were requested, and the 2 columns correspond to the x then y (i.e. lon then lat) coordinates.

- **Note** *(Rows are sorted by index number of the Individuals,)* – regardless of input index order.

get_e**(spp=0, lyr_num=None, individs=None)**

Returns the current environmental values of any or all Individuals in a Species, for one or all Landscape Layers

**Parameters**

- **spp**(int, str, optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **individs** *(iterable collection of ints, optional, default: None)* – If provided, indicates the integer indices of the Individuals whose environmental values should be returned. If None, all Individuals’ environmental values will be returned.

- **lyr_num**(int, default=None) – If provided, indicates the Layer for which values should be returned. If None, values will be returned for all Layers.

**Returns**

Returns an nxl numpy array, where n is the number of Individuals whose coordinates were requested, and l is the number of Layers requested, ordered by increasing Layer number.

Note: Rows are sorted by index number of the Individuals, regardless of input index order.

Return type **np.array**

get_fitness**(spp=0, trt=None, individs=None)**

Returns the current fitness of any or all Individuals in a Species, for one or all Traits

**Parameters**

- **spp**(int, str, optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **individs** *(iterable collection of ints, optional, default: None)* – If provided, indicates the integer indices of the Individuals whose fitness should be returned. If None, all Individuals’ fitness values will be returned.

- **trt**(int, str, optional, default: 0) – A reference to the Trait for which the GEA should be run. Can be either the Trait’s index number (i.e. its integer key in the GenomicArchitecture’s traits dict), or its name (as a character string).

**Returns**

Returns an nxt numpy array, where n is the number of Individuals whose coordinates were requested, and t is the number of Traits requested.

Note: Rows are sorted by index number of the Individuals, regardless of input index order.

Return type **np.array**
**get_genotypes** *(spp=0, loci=None, individs=None, biallelic=False)*

Returns the genotypes of any or all Individuals in a Species and for any or all loci.

**Parameters**

- **spp** *(int, str, optional, default: 0)* – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **loci** *(iterable collection of ints, optional, default: None)* – If provided, indicates the loci for which genotypes should be returned. If None, genotypes will be returned for all loci.

- **individs** *(iterable collection of ints, optional, default: None)* – If provided, indicates the integer indices of the only Individuals for whom genotypes should be returned. If None, all Individuals’ genotypes will be returned.

- **biallelic** *(bool, optional, default: False)* – If False, results are returned as nxL numpy arrays, where n is the number of Individuals requested and L is the number of loci requested. If True, results are returned as nxLx2 numpy arrays, where the 2 dimensions on the third axis represent the two haploid genotypes of each diploid Individual.

**Returns**

Returns a numpy array containing the requested genotypes. The shape of the array depends on the number of Individuals and loci and the return format requested. The first and second axes represent the Individuals and loci, returned in sorted numerical order. The optional third axis represents ploidy, if biallelic genotypes are requested. (See explanation of the argument ‘biallelic’, above, for more detail.)

Note: All axes are sorted by index number of the object they represent (Individuals, Loci), regardless of input index order.

**Return type** *np.array*

**get_random_individs** *(n, spp=0)*

Returns a size-n sample of index numbers for random individuals in a Species.

**Parameters**

- **n** *(int)* – The number of random Individuals to sample

- **spp** *(int, str, optional, default: 0)* – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**Returns**

Returns a length-n 1d numpy array containing the sorted index numbers of the sampled Individuals.

**Return type** *np.array*

**get_tree_sequence** *(spp=0)*

Returns the tskit.TreeSequence object for the indicated Species.

**Parameters**

- **spp** *(int, str, optional, default: 0)* – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**get_x** *(spp=0, individs=None)*

Returns the x coordinates of any or all Individuals in a Species.

**Parameters**
• **spp**((int, str), optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

• **individs** (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the Individuals whose coordinates should be returned. If None, all Individuals’ coordinates will be returned.

**Returns**

• np.array – Returns an nx1 numpy array, where n is the number of Individuals whose coordinates were requested.

• **Note** (Rows are sorted by index number of the Individuals,) – regardless of input index order.

---

**get_y**(spp=0, individs=None)

Returns the y coordinates of any or all Individuals in a Species

**Parameters**

• **spp**((int, str), optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

• **individs** (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the Individuals whose coordinates should be returned. If None, all Individuals’ coordinates will be returned.

**Returns**

• np.array – Returns an nx1 numpy array, where n is the number of Individuals whose coordinates were requested.

• **Note** (Rows are sorted by index number of the Individuals,) – regardless of input index order.

---

**get_z**(spp=0, trt=None, individs=None)

Returns the phenotypic values of any or all Individuals in a Species, for one or all Traits

**Parameters**

• **spp**((int, str), optional, default: 0) – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

• **individs** (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the Individuals whose phenotypes should be returned. If None, all Individuals’ phenotypes will be returned.

• **trt**((int, str), optional, default: 0) – A reference to the Trait for which the GEA should be run. Can be either the Trait’s index number (i.e. its integer key in the GenomicArchitecture’s traits dict), or its name (as a character string).

**Returns**

Returns an ntx numpy array, where n is the number of Individuals whose coordinates were requested, and t is the number of Traits requested.

**Note**: Rows are sorted by index number of the Individuals, regardless of input index order.

**Return type** np.array
plot (spp=None, lyr=None, hide_land=False, individs=None, text=False, color='black', edge_color='face', text_color='black', cbar=True, size=25, text_size=9, land_cmap=None, pt_cmap=None, alpha=False, zoom_width=None, x=None, y=None, vmin=None, vmax=None, ticks=False, mask_rast=None, animate=False)

Create a basic plot of the Model in its current state.

Plot the current state of the model, with any number of overlain Landscape Layers, and with or without points representing the Individuals in a Species.

Parameters

- **spp** ((int, str), optional, default: None) – A reference to the Species whose Individuals should be scattered on the plot. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string). If None, will cause only the Landscape to be plotted.

- **lyr** ((int, str), optional, default: None) – A reference to the Layer whose raster should be plotted. Can be either the Layer’s index number (i.e. its integer key in the Landscape dict), or its name (as a character string). Defaults to None, which will cause all Layers to be plotted as an overlay of transparent rasters, each with a different colormap.

- **hide_land** (bool, optional, default: False) – If True, the Landscape will be ‘hidden’, i.e. the plot will only show a scatter of the plotted Individuals. Defaults to False.

- **individs** (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the only Individuals to be plotted. If None, all Individuals will be plotted.

- **text** (bool, optional, default: False) – If True, each Individual’s index number will be displayed next to it. Can be useful for model introspection.

- **color** (valid mpl.plt color value, optional, default: ‘black’) – Face color for the points in the Individual scatter. Passed to the c argument of matplotlib.pyplot.scatter.

- **edge_color** (valid mpl.plt color value, optional, default: ‘face’) – Edge color for the points in the Individual scatter. If ‘face’, will always match the face color (i.e. the color provided to color). Passed to the edgecolor argument of matplotlib.pyplot.scatter.

- **text_color** (valid mpl.plt color value, default 'black') – Color for the plotted text. (Will only be used if the text argument is True.) Passed to the color argument of matplotlib.pyplot.text.

- **cbar** (bool, optional, default: True) – If True, a colorbar will be included, depicting the mapping of color onto the Landscape’s environmental values.

- **size** (scalar or array_like, optional, default: 25) – Size of the scatter points. Passed to the s argument of matplotlib.pyplot.scatter.

- **text_size** ((size in points, valid string), optional, default: 9) – Text size. (Will only be used if text is True.) Can be expressed as a numeric size in points, or as any string that is a valid size argument for matplotlib.text.Text.

- **land_cmap** ((valid string, None), optional, default: None) – Colormap to use for plotting Layer rasters. If None, will default to the colormap automatically assigned to the Layer by Geonomics based on the Layer’s index number in the Landscape dict. Can be passed any string that references a colormap in matplotlib.pyplot.cm.
• **pt_cmap** *(valid string, None), optional, default: None)* – Colormap to use for plotting points. (Will only be used if the Individual scatter is being plotted, and if the value provided to color is an ordered collection of values to which to map the Individuals’ colors.) If None, no colormap will be used. Can be passed any string that references a colormap in matplotlib.pyplot.cm, as well as an matplotlib.colors.LinearSegmentedColormap instance.

• **alpha** *(scalar, bool, None), optional, default: None)* – The transparency level of the points. If scalar is passed, it is passed on to the alpha value fed to matplotlib.pyplot.scatter. If True, alpha value will be set to 0.6. If False or None, points will be opaque.

• **zoom_width** *(scalar, None), optional, default: None)* – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.

• **x,y** *(scalar, None), default: None)* – The x and y coordinates of the center of the resulting plot. Only used if zoom_width is not None.

• **vmax**(vmin,)* – Used to normalize the luminance data that determine how colors are colormapped onto Individuals’ point colors. Passed to the vmin and vmax arguments of matplotlib.pyplot.scatter.

• **ticks** *(bool, optional, default: False)* – If True, x- and y-axis ticks will be added to the plot.

• **mask_rast** *(array_like, optional default: None)* – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

• **animate** *(bool, optional, default: False)* – If True, this function will return a list of the Individuals’ points, as a matplotlib.collections.PathCollection. This is used internally by the Model.walk method, to create animated simulation plots.

**Returns** Returns no output (unless animate is passed True, in which case a PathCollection of the Individuals’ points is returned, for internal use by Model.walk, to create animated simulations)

**Return type** {None, matplotlib.collections.PathCollection}

**Notes**

* For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

**plot_allele_frequencies**(spp=0)
Plot a Species’ current allele frequencies for all alleles

For the Species indicated, create a plot showing, across all loci, the current allele frequencies (in solid blue) versus the starting frequencies (in dashed red).

**Parameters** **spp** *(int, str), optional, default: 0)* – A reference to the Species whose allele frequencies will be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**Returns** Returns no output.

**Return type** None
**plot_demographic_changes** *(spp=0)*

TEMPORARILY OUT OF ORDER. NEEDS TO BE DEBUGGED.

**plot_demographic_pyramid** *(spp=0)*

Plot a demographic pyramid for the chosen Species

Plot a paired, horizontal bar plot of the chosen Species’ count of Individuals of each age.

**Parameters**

- **spp** (*{int, str}, optional, default: 0*) – A reference to the Species whose demographic pyramid will be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**Returns**

Returns no output

**Return type** None

**plot_density** *(spp=0, normalize=False, individs=None, text=False, color='black', edge_color='face', text_color='black', size=25, text_size=9, alpha=0.5, zoom_width=None, x=None, y=None, ticks=None, mask_rast=None)*

Plot the population-density raster for a given Species.

Plot the population density of a Species, as estimated by the Species’ _DensityGridStack_.

**Parameters**

- **spp** (*{int, str}, optional, default: 0*) – A reference to the Species whose Individuals should be scattered on the plot. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **normalize** (*bool, optional, default: False*) – If True, the population-density values of the output raster will be normalized to $0 \leq density \leq 1$. Otherwise, actual estimates (individuals per cell) are returned.

- **individs** (*iterable collection of ints, optional, default: None*) – If provided, indicates the integer indices of the only Individuals to be plotted. If None, all Individuals will be plotted.

- **text** (*bool, optional, default: False*) – If True, each Individual’s index number will be displayed next to it. Can be useful for model introspection.

- **color** (*valid mpl.plt color value, optional, default: ‘black’*) – Face color for the points in the Individual scatter. Passed to the *c* argument of matplotlib.pyplot.scatter.

- **edge_color** (*valid mpl.plt color value, optional, default: ‘face’*) – Edge color for the points in the Individual scatter. If ‘face’, will always match the face color (i.e. the color provided to *color*). Passed to the *edgelcolor* argument of matplotlib.pyplot.scatter.

- **text_color** (*valid mpl.plt color value, default ‘black’*) – Color for the plotted text. (Will only be used if the *text* argument is True.) Passed to the *color* argument of matplotlib.pyplot.text.

- **size** (*scalar or array_like, optional, default: 25*) – Size of the scatter points. Passed to the *s* argument of matplotlib.pyplot.scatter.

- **text_size** (*{size in points, valid string}, optional, default: 9*) – Text size. (Will only be used if *text* is True.) Can be expressed as a numeric size in points, or as any string that is a valid size argument for matplotlib.text.Text.

- **alpha** (*{scalar, bool, None}, optional, default: None*) – The transparency level of the points. If scalar is passed, it is passed on to the *alpha* value
fed to matplotlib.pyplot.scatter. If True, alpha value will be set to 0.6. If False or None, points will be opaque.

- **zoom_width** ((scalar, None), optional, default: None) – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.

- **x, y** ((scalar, None), default: None) – The x and y coordinates of the center of the resulting plot. Only used if **zoom_width** is not None.

- **ticks** (bool, optional, default: False) – If True, x- and y-axis ticks will be added to the plot.

- **mask_rast** (array_like, optional, default: None) – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

**Returns** Returns no output

**Return type** None

**Notes**

- For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

**plot_dispersal_surface** (spp=0, style='hist', x=None, y=None, zoom_width=None, scale_fact=4.5, color='black', cbar=True, ticks=None, cmap='plasma', mask_rast=None)

Create any of 4 types of plots to summarize a Species’ DispersalSurface

Plot a Species’ DispersalSurface with either circularized histograms, a circle of directional draws, or a vector on each cell, to visualize the von Mises distributions of each cell on the MovementSurface. Or plot a plain histogram of the von Mises distribution at a given cell.

**Parameters**

- **spp** ((int, str), optional, default: 0) – A reference to the Species whose DispersalSurface should be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **style** (str, optional, default: ‘hist’) – Determines the style of plot produced. There are four options:
  - **hist**: Plot a classic histogram approximating the von Mises distribution at the cell indicated by position x,y. (Note that x and y must be provided for this style.)
  - **chist**: Plot a circular histogram approximating the von Mises distribution at the cell indicated by position x,y; plot will be drawn inside the chosen cell on the Dispersal-Surface raster.
  - **cdraws**: Plot points on the unit circle, whose locations were drawn from the von Mises distribution at the cell indicated by position x,y; plot will be drawn inside the chosen cell on the DispersalSurface raster.
  - **vect**: Inside each cell of the DispersalSurface raster, plot the mean direction vector of directions drawn from that cell’s von Mises distribution.
• **color** *(valid mpl.plt color value, optional, default: 'black')* – Color to use for the plot elements. Passed to the `c` argument of matplotlib.pyplot.hist, matplotlib.pyplot.plot, matplotlib.pyplot.scatter, or matplotlib.pyplot.arrow, depending on `style`.

• **cbar** *(bool, optional, default: True)* – If True, a colorbar will be included, depicting the mapping of color onto the ModelSurface’s environmental values.

• **cmap** *(valid string, None), optional, default: 'plasma')* – Colormap to use for plotting the DispersalSurface. If None, will default to ‘plasma’, a common colormap used to plot resistance surfaces. Can be passed any string that references a colormap in matplotlib.pyplot.cm.

• **zoom_width** *(scalar, None), optional, default: None)* – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.

• **x,y** *(scalar, None), default: None)* – The x and y coordinates of the center of the resulting plot. Only used if `zoom_width` is not None.

• **mask_rast** *(array_like, optional default: None)* – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

**Returns** Returns no output

**Return type** None

**Notes**

• Not valid for Species without a DispersalSurface

• For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

**plot_example_recombinant_genome** *(spp=0)*

Plot an example recombinant genome for the chosen Species

Plot a vertical image depicting an example recombinant genome for the chosen Species. Each gamete (i.e. vertical half) is colored using a different colormap, and for each one alternating bands of color indicate breakpoints where recombination occurred between the two parental chromatids (or haploid chromatid sets, for multi-chromosome models). The resulting plot can be useful for introspecting or exploring a model, because each plot will be unique, and will be a result of the array of interlocus recombination rates defined in the Species’ GenomicArchitecture.

**Parameters**

• **spp** *(int, str), optional, default: 0)* – A reference to the Species for which an example recombinant genome should be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**Returns** Returns no output

**Return type** None

**Notes**

• Not valid for Species without genomes.
The function `plot_fitness` is used to create a scatter plot of the Individuals in a Species, on top of any Landscape Layer, and color the points by their fitnesses. Fitness values can be either overall fitness (i.e. calculated across all traits), or can be fitnesses for a certain Trait.

Parameters:

- **spp**(int, str, optional, default: 0) – A reference to the Species whose Individuals should be scattered on the plot. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string). If None, will cause only the Landscape to be plotted.

- **trt**(int, str, optional, default: None) – A reference to the Trait for which fitness should be calculated and used to color Individuals in the plot. Can be either the Trait’s index number (i.e. its integer key in the GenomicArchitecture’s traits dict), or its name (as a character string). If None, Individuals’ overall fitnesses will be calculated instead.

- **lyr**(int, str, optional, default: None) – A reference to the Layer whose raster should be plotted. Can be either the Layer’s index number (i.e. its integer key in the Landscape dict), or its name (as a character string). Defaults to None, which will either cause only the Layer associated with the chosen Trait to be plotted, if `trt` is not None; or else will cause all Layers to be plotted as an overlay of transparent rasters, each with a different colormap, if `trt` is None.

- **individs**(iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the only Individuals to be plotted. If None, all Individuals will be plotted.

- **text**(bool, optional, default: False) – If True, each Individual’s index number will be displayed next to it. Can be useful for model introspection.

- **phenotype_text**(bool, optional, default: False) – If True, each Individual’s phenotype will be displayed next to it. Can be useful for model introspection.

- **fitness_text**(bool, optional, default: False) – If True, each Individual’s calculated fitness value will be displayed next to it. Can be useful for model introspection.

- **edge_color**(valid mpl.pyplot color value, optional, default: ‘black’) – Edge color for the points in the Individual scatter. If ‘face’, will always match the face color (i.e. the color provided to `color`). Passed to the `edgecolor` argument of matplotlib.pyplot.scatter.

- **text_color**(valid mpl.pyplot color value, default ‘black’) – Color for the plotted text. (Will only be used if the `text` argument is True.) Passed to the `color` argument of matplotlib.pyplot.text.

- **fitness_text_color**(phenotype_text_color,) –

- **'black'(default)** – Color for the plotted phenotype-value or fitness-value text. (Will only be used if the `phenotype_text` or `fitness_text` arguments are True.) Passed to the `color` argument of matplotlib.pyplot.text.
Plot a Species’ Individuals, colored by their genotypes for a locus.

Create a scatter plot of the Individuals in a Species, on top of any Landscape Layer, and color the points by their genotypes for the chosen locus.

**Parameters**

- `spp=0`, `locus=0`, `lyr=None`, `by_dominance=False`, `individs=None`, `text=False`, `edge_color='black'`, `text_color='black'`, `cbar=True`, `size=25`, `text_size=9`, `alpha=1`, `zoom_width=None`, `x=None`, `y=None`, `ticks=None`, `mask_rast=None`
• **spp** *(int, str, optional, default: 0)* – A reference to the Species whose Individuals should be scattered on the plot. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

• **locus** *(int, optional, default: 0)* – The index of the locus for which Individuals’ genotypes should be depicted. Must be 0 ≤ val ≤ (genome length - 1)

• **lyr** *(int, str, optional, default: None)* – A reference to the Layer whose raster should be plotted. Can be either the Layer’s index number (i.e. its integer key in the Landscape dict), or its name (as a character string). Defaults to None, which will cause all Layers to be plotted as an overlay of transparent rasters, each with a different colormap.

• **by_dominance** *(bool, optional, default: False)* – If False, the Individuals will be colored by their actual genotypes (i.e. a 0|0 homozygote will be depicted as 0, a 1|1 homozygote as 1, and a 0|1 heterozygote as 0.5). If True, then the dominance value for the chosen locus will be used to determine the ‘effective genotype’, which will instead be displayed as an Individual’s color. In this case, 0|1 heterozygotes will be depicted as 0.5’s for loci with codominance (i.e. dominance values of 0), but will be depicted as 1’s for loci with 1-allele dominance (i.e. dominance values of 1). The dominance values for all loci in the genome are set in a Species’ GenomicArchitecture object. For more detail, see the “Genomes, GenomicArchitecture, and Traits” subsection of the “Data Structures” section of the online documentation.

• **individs** *(iterable collection of ints, optional, default: None)* – If provided, indicates the integer indices of the only Individuals to be plotted. If None, all Individuals will be plotted.

• **text** *(bool, optional, default: False)* – If True, each Individual’s index number will be displayed next to it. Can be useful for model introspection.

• **edge_color** *(valid mpl.pyplot color value, optional, default: 'face')* – Edge color for the points in the Individual scatter. If ‘face’, will always match the face color (i.e. the color provided to color). Passed to the edgecolor argument of matplotlib.pyplot.scatter.

• **text_color** *(valid mpl.pyplot color value, default 'black')* – Color for the plotted text. (Will only be used if the text argument is True.) Passed to the color argument of matplotlib.pyplot.text.

• **cbar** *(bool, optional, default: True)* – If True, a colorbar will be included, depicting the mapping of color onto the Landscape’s environmental values.

• **size** *(scalar or array_like, optional, default: 25)* – Size of the scatter points. Passed to the s argument of matplotlib.pyplot.scatter.

• **text_size** *(size in points, valid string, optional, default: 9)* – Text size. (Will only be used if text is True.) Can be expressed as a numeric size in points, or as any string that is a valid size argument for matplotlib.text.Text.

• **alpha** *(scalar, bool, None), optional, default: None)* – The transparency level of the points. If scalar is passed, it is passed on to the alpha value fed to matplotlib.pyplot.scatter. If True, alpha value will be set to 0.6. If False or None, points will be opaque.

• **zoom_width** *(scalar, None), optional, default: None)* – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.
• **x, y**({*scalar, None*}, default: *None*) – The x and y coordinates of the center of the resulting plot. Only used if **zoom_width** is not None.

• **ticks**(*bool, optional, default: False*) – If True, x- and y-axis ticks will be added to the plot.

• **mask_rast**(*array_like, optional default: None*) – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

**Returns**  Returns no output

**Return type**  None

**Notes**

- For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

### plot_hist_fitness(**spp=0**)

Plot a Species

For the Species indicated, create a histogram of Individuals’ current fitness values.

**Parameters**  **spp** (*{int, str}, optional, default: 0*) – A reference to the Species whose Individuals’ fitness will be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

**Returns**  Returns no output.

**Return type**  None

### plot_movement_surface(**spp=0**, **style='hist'**, **color='black'**, **cbar=True**, **ticks=None**, **cmap='plasma'**, **zoom_width=None**, **x=None**, **y=None**, **scale_fact=4.5**, **mask_rast=None**)

Create any of 4 types of plots to summarize a Species’ MovementSurface

Plot a Species’ MovementSurface with either circularized histograms, a circle of directional draws, or a vector on each cell, to visualize the von Mises distributions of each cell on the MovementSurface. Or plot a plain histogram of the von Mises distribution at a given cell.

**Parameters**

- **spp**(*{int, str}, optional, default: 0*) – A reference to the Species whose MovementSurface should be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **style**(*str, optional, default: 'hist'* )–

  Determines the style of plot produced. There are four options:

  - **'hist'**: Plot a classic histogram approximating the von Mises distribution at the cell indicated by position x,y. (Note that x and y must be provided for this style.)

  - **'chist'**: Plot a circular histogram approximating the von Mises distribution at the cell indicated by position x,y; plot will be drawn inside the chosen cell on the MovementSurface raster.
'cdraws': Plot points on the unit circle, whose locations were drawn from the von Mises distribution at the cell indicated by position x,y; plot will be drawn inside the chosen cell on the MovementSurface raster.

'vect': Inside each cell of the MovementSurface raster, plot the mean direction vector of directions drawn from that cell’s von Mises distribution.

• color (valid mpl.plt color value, optional, default: 'black') – Color to use for the plot elements. Passed to the c argument of matplotlib.pyplot.hist, matplotlib.pyplot.plot, matplotlib.pyplot.scatter, or matplotlib.pyplot.arrow, depending on style.

• cbar (bool, optional, default: True) – If True, a colorbar will be included, depicting the mapping of color onto the MovementSurface’s environmental values.

• cmap((valid string, None), optional, default: 'plasma') – Colormap to use for plotting the MovementSurface. If None, will default to 'plasma', a common colormap used to plot resistance surfaces. Can be passed any string that references a colormap in matplotlib.pyplot.cm.

• zoom_width ((scalar, None), optional, default: None) – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.

• x, y ((scalar, None), default: None) – The x and y coordinates of the center of the resulting plot. Only used if zoom_width is not None.

• mask_rast (array_like, optional default: None) – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

Returns Returns no output

Return type None

Notes

• Not valid for Species without a DisersalSurface

• For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

plot_phenotype (spp=0, trt=0, lyr=None, indivs=None, text=False, edge_color='black', text_color='black', cbar=True, size=25, text_size=9, alpha=1, zoom_width=None, x=None, y=None, ticks=None, mask_rast=None, animate=False)

Plot a Species’ Individuals, colored by their phenotypes for a Trait

Create a scatter plot of the Individuals in a Species, on top of any Landscape Layer, and color the points by their phenotypes for the chosen Trait.

Parameters

• spp((int, str), optional, default: 0) – A reference to the Species whose Individuals should be scattered on the plot. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).
• `trt ((int, str), optional, default: 0) – A reference to the Trait whose phenotypes should be used to color Individuals in the plot. Can be either the Trait’s index number (i.e. its integer key in the GenomicArchitecture’s traits dict), or its name (as a character string).

• `lyr ((int, str), optional, default: None) – A reference to the Layer whose raster should be plotted. Can be either the Layer’s index number (i.e. its integer key in the Landscape dict), or its name (as a character string). Defaults to None, which will cause only the Layer associated with the chosen Trait to be plotted.

• `individs (iterable collection of ints, optional, default: None) – If provided, indicates the integer indices of the only Individuals to be plotted. If None, all Individuals will be plotted.

• `text (bool, optional, default: False) – If True, each Individual’s index number will be displayed next to it. Can be useful for model introspection.

• `edge_color (valid mpl.plt color value, optional, default: 'face') – Edge color for the points in the Individual scatter. If ‘face’, will always match the face color (i.e. the color provided to color). Passed to the edgecolor argument of matplotlib.pyplot.scatter.

• `text_color (valid mpl.plt color value, default 'black') – Color for the plotted text. (Will only be used if the text argument is True.) Passed to the color argument of matplotlib.pyplot.text.

• `cbar (bool, optional, default: True) – If True, a colorbar will be included, depicting the mapping of color onto the Landscape’s environmental values.

• `size (scalar or array_like, optional, default: 25) – Size of the scatter points. Passed to the s argument of matplotlib.pyplot.scatter.

• `text_size ((size in points, valid string), optional, default: 9) – Text size. (Will only be used if text is True.) Can be expressed as a numeric size in points, or as any string that is a valid size argument for matplotlib.text.Text.

• `alpha ((scalar, bool, None), optional, default: None) – The transparency level of the points. If scalar is passed, it is passed on to the alpha value fed to matplotlib.pyplot.scatter. If True, alpha value will be set to 0.6. If False or None, points will be opaque.

• `zoom_width (scalar, None), optional, default: None) – Width, in raster cells, of the window to which to zoom to resulting plot. If None, plot shows full Landscape.

• `x, y (scalar, None), default: None) – The x and y coordinates of the center of the resulting plot. Only used if zoom_width is not None.

• `ticks (bool, optional, default: False) – If True, x- and y-axis ticks will be added to the plot.

• `mask_rast (array_like, optional default: None) – An array to use to mask values in the Layer(s) being plotted. All np.nan values in the array will be plotted as light gray, instead of the color that would otherwise map to their environmental values in their raster(s). The array must be of the same x,y dimensions as the Layer(s) being plotted.

• `animate (bool, optional, default: False) – If True, this function will return a list of the Individuals’ points, as a matplotlib.collections.PathCollection. This is used internally by the Model.walk method, to create animated simulation plots.
**Returns** Returns no output (unless animate is passed True, in which case a PathCollection of the Individuals’ points is returned, for internal use by Model.walk, to create animated simulations)

**Return type** `{None, matplotlib.collections.PathCollection}

**Notes**

- For more detail on plotting parameters, see the documentation for matplotlib, matplotlib.pyplot.pcolormesh, matplotlib.pyplot.scatter, and matplotlib.pyplot.text.

```python
plot_pop_growth(spp=0, expected=True, actual=True, expected_color='red', actual_color='blue')
```

Plot the chosen Species’ population over time

Plot a line plot of the Species’ population over model time (in blue), as well as the expected population-size trend (based on summing the Species’ carrying-capacity raster and using that total expected population size to solve the logistic growth equation; in red).

**Parameters**

- `spp((int, str), optional, default: 0)` – A reference to the Species whose population growth will be plotted. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- `expected(bool, default: True)` – Whether or not to plot the line showing the expected (under the logistic growth model) trajectory of population growth, in red.

- `actual(bool, default: True)` – Whether or not to plot the line showing the actual trajectory of population growth, in blue.

**Returns** Returns no output

**Return type** None

```python
plot_stat(stat, spp=0)
```

TEMPORARILY OUT OF ORDER. NEEDS TO BE DEBUGGED.

```python
run(verbos=False)
```

Run a Model.

Run the Model for the number of iterations that was stipulated in the parameters file used to create it (which is saved as the n_its attribute). During each iteration, the model will burn in for at least the stipulated number of burn-in timesteps (the burn_T attribute), then will run for the stipulated number of main timesteps (the ‘T’ attribute). If stats and/or data are going to be collected (also stipulated by the parameters file used to create the Model), then the output files for each iteration will be saved in a separate subdirectory, containing further subdirectories for each Species.

**Parameters**

- `verbose(bool, optional)` – Whether or not to run the Model should provide written output. If True, formatted messages will be printed to STDOUT at each timestep (which could be piped to a log file, if desired).

**Returns** `out` – Returns no output. Writes information to STDOUT if verbose is True. Writes data and statistics to file if so parameterized.

**Return type** None
Examples

We can create and run the default model (as long as there is no “GNX_params_<...>.py” file in the current working directory before gnx.make_parameters_file is called).

```python
>>> gnx.make_parameters_file()
>>> mod = gnx.make_model()
NOTE: Using the following file, in the current working directory to create the Model object:
   GNX_params_21-01-2020_17:22:08.py
>>> mod.run(verbos = True)
Running model "GNX_params_21-01-2020_17:22:08"...
```

```plaintext
Setting up iteration 0...
Creating the burn-in function queue...
Creating the main function queue...
Running burn-in, iteration 0...
burn: it=0: t=0
   species: spp_0 N=250 (births=35 deaths=214)
...............................................................................
burn: it=0: t=1
   species: spp_0 N=250 (births=35 deaths=214)
...............................................................................
burn: it=0: t=2
   species: spp_0 N=250 (births=35 deaths=214)
...............................................................................
```

```python
run_gea (method='CCA', spp=0, trt=0, plot=True, plot_sd=True, scale=3, sd=3)
Runs one of a number of available GEA models, returning the results and optionally plotting them.
```

Parameters

- **method** *(str, optional, default: "CCA")* – A string indicating which type of GEA model to run. NOTE: Currently, only CCA (default) is available.
- **spp** *(int, str, optional, default: 0)* – A reference to the Species for which the GEA should be run. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).
- **trt** *(int, str, optional, default: 0)* – A reference to the Trait for which the GEA should be run. Can be either the Trait’s index number (i.e. its integer key in the GenomicArchitecture’s traits dict), or its name (as a character string).
- **plot** *(bool, optional, default: True)* – Whether or not the GEA model should be plotted.
- **plot_sd** *(bool, optional, default: True)* – Whether or not a standard-deviation ellipse should be added to the GEA’s plot. (Only used if
plot==True and method=="CCA".)

• **scale** ([int, float], optional, default: 3) – The scaling factor to use when plotting the GEA results. (Only used if plot==True.)

• **sd** ([int, float], optional, default: 3) – The number of standard deviations to use when plotting the standard-deviation ellipse. (Only used if plot==True and method=="CCA".)

Returns Returns a dict containing the labeled (as keys) output data structures (as values) of the chosen method of GEA.

Return type dict

### walk

(T=1, mode='main', verbose=True, animate=False)

Walk through a Model (i.e. run it for a certain number of timesteps).

This function will run a Model for a specific number of timesteps, either in ‘burn’ or ‘main’ mode. It is designed to help users to try out, explore, and introspect their Models in an interactive Python session. (For best results, we strongly recommend this be done with the iPython <https://ipython.org> shell.)

Parameters

• **T** ([int, optional] – The number of timesteps for which to run the Model. (If in ‘main’ mode, the Model will run for precisely this many timesteps. If in ‘burn’ mode, it will run either until stationarity statistics indicate that it has burned in, or for this many timesteps, whichever occurs first.) Defaults to a single timestep.

• **mode** ([{'burn', 'main'}, optional] – The mode in which to run the Model. If ‘burn’, it will run in burn-in mode. This entails the following:

  – Genomes will not yet be assigned, so genomic phenomena such as crossing-over and natural selection will not occur;

  – Any scheduled landscape and demographic changes will not occur;

  – Stationarity statistics will be checked at the end of each timestep, to determine if the Model has burned in (in which case the model will stop running regardless of timestep, genomes will be randomly assigned to all individuals, and the Model will be deemed ready to run in main mode)

If ‘main’, it will run in main mode (i.e. all functionalities will be used. (Note that for a Model to run in main mode it must first have been burned in.)

• **verbose** (bool, optional) – Whether or not to run the Model should provide written output. If True, formatted messages will be printed to STDOUT at each timestep. Defaults to True for Model.walk.

• **animate** ([tuple of ints, bool], optional) – If a length-2 tuple of integers is provided, the Species indicated by the first number will be plotted on the Landscape Layer indicated by the second number at each timestep, in a dynamically updating plot. If a length-3 tuple of integers in provided, the Species indicated by the first number will be plotted on the Landscape Layer indicated by the second number at each timestep, colored by the phenotypes of the trait indicated by the third number. If just True, the first Species (index 0) will be plotted on top of a transparent stack of all Landscape Layers. (Note that this will slow down the execution of a model a bit, because the plot will pause for 0.1 seconds after each timestep.)

Returns **out** – Returns no output. Writes information to STDOUT if verbose is True. Writes data and statistics to file if so parameterized.

Return type None
**Raises** `ValueError` – If the user attempts to run the Model in ‘main’ mode before it has been burned in.

**Examples**

We can create and walk the default model (as long as there is no “GNX_params_<...>py” file in the current working directory before `gnx.make_parameters_file` is called).

```python
>>> gnx.make_parameters_file()
>>> mod = gnx.make_model()
NOTE: Using the following file, in the current working directory to create the Model object:
    GNX_params_21-01-2020_17:22:08.py
>>> # run the burn-in until it is complete
>>> mod.walk(T=1000, mode='burn')
Running model "GNX_params_21-01-2020_17:22:08"...

Setting up iteration 0...
Creating the burn-in function queue...
Creating the main function queue...
Running burn-in, iteration 0...
burn: it=0: t=0
    species: spp_0 N=250 (births=35 deaths=214)
    
-burn: it=0: t=1
    species: spp_0 N=250 (births=35 deaths=214)
    
-burn: it=0: t=2
    species: spp_0 N=250 (births=35 deaths=214)
    
Assigning genomes for species "spp_0"...
Burn-in complete.

>>> # now run in main mode for 50 timesteps
>>> mod.walk(T=50, mode='main')
main: it=1: t=0
    species: spp_0 N=131 (births=35 deaths=214)
    
-main: it=1: t=1
    species: spp_0 N=129 (births=35 deaths=214)
    
-main: it=1: t=2
```

(continues on next page)
write_gendata (filepath, spp=0, n=None, include_fixed_sites=True)

A convenience function, designed for interactive use, that writes genetic data for a sample of n Individuals to the given filepath

Parameters

- **filepath** (str) – The path and filename indicating the destination of the output file. Output format (‘VCF’, ‘FASTA’) will be determined by the filepath extension (‘.vcf’, ‘.fasta’).

- **spp** (int, str), optional, default: 0 – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **n** (int, None), optional, default: None – The number of individuals of randomly sample. If None, output file will contain the entire current population.

- **include_fixed_sites** (bool, optional, default: True) – A flag indicating whether or not to include fixed sites. Only valid for VCF files. Defaults to True.

Notes

This function is only designed for use with small models or in interactive mode. For simulations being run to generate datasets for research use, it is preferable to generate a parameters file containing a section for data section (gnx.make_parameters_file(…, data=True)), then parameterize data collection there.

write_geodata (filepath, spp=0, n=None)

A convenience function, designed for interactive use, that writes tabular geodata for a sample of n Individuals to the given filepath

Parameters

- **filepath** (str) – The path and filename indicating the destination of the output file. Output format (‘CSV’, ‘Shapefile’, ‘GeoJSON’) will be determined by the filepath extension (‘.csv’, ‘.shp’, ‘.json’).

- **spp** (int, str), optional, default: 0 – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).

- **n** (int, None), optional, default: None – The number of individuals of randomly sample. If None, output file will contain the entire current population.
Notes

This function is only designed for use with small models or in interactive mode. For simulations being run to generate datasets for research use, it is preferable to generate a parameters file containing a section for data section (`gnx.make_parameters_file(., data=True)`), then parameterize data collection there.

write_tskit_table_collection/file_basename, spp=0, sep=',')

A convenience function, designed for interactive use, that writes the tskit.TableCollection containing the spatial pedigree into CSV tables

Parameters

- **file_basename (str)** – The path and file basename to use for the resulting table files. Each file will be appended with a capitalized suffix indicating the table it contains (NODES, EDGES, SITES, MUTATIONS, INDIVIDUALS).
- **spp ((int, str), optional, default: 0)** – A reference to the Species for which values should be returned. Can be either the Species’ index number (i.e. its integer key in the Community dict), or its name (as a character string).
- **sep (str, optional, default: ',')** – Character separator to use in the output CSV

Notes

This function is only designed for use with small models or in interactive mode. For simulations being run to generate datasets for research use, it is preferable to generate a parameters file containing a section for data section (`gnx.make_parameters_file(., data=True)`), then parameterize data collection there.
class geonomics.structs.community.Community(land, spps)
    Bases: dict
    Representation of a community (i.e. a collection of Species objects)
    Structured as a serial integer-keyed dict of Species objects.
    Because the Community class inherits from dict, Species can be indexed out using the index-number keys (e.g. mod.comm[<idx>]).
    The Community is stored as the ‘mod.comm’ attribute of its corresponding Model object.
    NOTE: There is currently no special functionality implemented for multi-species models (e.g. species interactions), such that this serves as nothing more than a way to synchronously simulate two species within the same model. However, this class is intended as a scaffold for use in possible future expansion of Geonomics to incorporate multi-species functionality.
    burned
        A bool flag indicating whether or not all Species within the Community have been burned in.
    n_spps
        The number of Species in the Community
    t
        The index of the last completed time step run for this Community. This attribute holds the value -1 until all Species in the Community have been burned in, at which point it begins incrementing, Pythonically, from 0 (such that a Community with t == 999 has been run for 1000 time steps).

geneomics.structs.genome module

Classes, associated methods, and supporting functions for all genomic components

class geonomics.structs.genome.GenomicArchitecture(dom, g_params, land, recomb_rates=None, recomb_positions=None)
    Bases: object
    Representation of the genomic architecture of a Species.
    Inheres to its Species as the Species.gen_arch attribute.
    NOTE that correspond to many of the following attributes.
    Type For more detail, see the documentation for the parameters
    delet_alpha_distr_scale
        The scale parameter of the Gamma distribution from which the effect sizes (i.e. alphas) of deleterious loci are drawn
    delet_alpha_distr_shape
        The shape parameter of the Gamma distribution from which the effect sizes (i.e. alphas) of deleterious loci are drawn
    delet_loci
        A 1d numpy array that tracks all deleterious loci
    delet_loc_idx
        A 1d numpy array containing the index positions within an Individual’s non-neutral genotypes object (attribute ‘g’ of an Individual) that correspond to the deleterious loci. (This is designed such that an Individual’s genotype for the deleterious locus number stored at GenomicArchitecture.delet_loci[i] can be
accessed by calling Individual.g[GenomicArchitecture.delet_loc_idx[i], :].) This method of tracking allows Individuals’ full genotypes to be stored in the tskit tables while they carry their non-neutral genotypes with them (a computational optimization).

**delet_loci_s**
A 1d numpy array containing the selection strengths of each of the deleterious loci. (Selection strengths are listed in an order corresponding to the order of GenomicArchitecture.delet_loci, such that the selection strength of the deleterious locus referenced at GenomicArchitecture.delet_loci[i] is stored at GenomicArchitecture.delet_loci_s[i].)

**dom**
Dominance values for all loci (stored as an L-length, 1d numpy array in which 0 indicates a codominant locus and 1 indicates a locus for which the ‘1’ allele is dominant)

**L**
Length of the simulated genome

**mu_delet**
Genome-wide deleterious mutation rate (expressed in mutations per locus per timestep). Deleterious mutations do not influence trait phenotypes, but instead are universally deleterious (to an extent determined by their effect sizes).

**mu_neut**
Genome-wide neutral mutation rate (expressed in mutations per locus per timestep).

**neut_loci**
A 1d numpy array that tracks all loci that do not influence the phenotypes of any traits.

**nonneut_loci**
A 1d numpy array that tracks all loci that either influence the phenotype of at least one trait or are deleterious.

**p**
Starting allele frequencies for all loci (stored as an L-length, 1d numpy array)

**pleiotropy**
A bool indicating whether or not to allow pleiotropy (i.e. whether or not to allow the same locus to subtend multiple traits)

**recombinations**
The Recombinations object, which contains the parameters and data necessary for simulation of recombination. Not intended for public use at this time.

**sex**
A bool indicating whether or not Individuals of this Species will be assigned a sex (0 for female, 1 for male)

**traits**
A dict containing all of a Species’ Trait objects, keyed by serial integers.

**x**
Ploidy NOTE: only diploidy is currently implemented, but this attribute serves as a placeholder for use in possible future generalization to arbitrary ploidy.)

```python
exception geonomics.structs.genome.MutationRateError
Bases: Exception
class geonomics.structs.genome.Recombinations(L, positions, n, r_distr_alpha, r_distr_beta, recomb_rates, jitter_breakpoints)
Bases: object
```
Container for the parameters and data necessary to simulate recombination.
Not intended for public use.

class geonomics.structs.genome.Trait(idx, name, phi, n_loci, mu, layer, alpha_distr_mu, alpha_distr_sigma, max_alpha_mag, gamma, univ_adv)

Bases: object

Representation of a single trait and its parameterization.

Trait objects are collected in a serial integer-keyed dict, one per trait that is parameterized for the corresponding Species, which inheres to the Species’ GenomicArchitecture object as the attribute GenomicArchitecture.traits, from which individual Traits can be indexed using their index-number keys (e.g. GenomicArchitecture.traits[<idx>]).

NOTE
that correspond to many of the following attributes.

Type For more detail, see the documentation for the parameters

alpha
A 1d numpy array containing the effect sizes of all loci subtending this trait. The value at index i represents the effect size of the locus at index i in the ‘loci’ attribute of this Trait.

alpha_distr_mu
Mean of the normal (Gaussian) distribution of effect sizes for this trait’s loci.

alpha_distr_sigma
Standard deviation of the normal (Gaussian) distribution of effect sizes for this trait’s loci.

gamma
Curvature of the fitness function.

idx
Index number of the trait (i.e. its key within the traits dict)

loc_idx
A 1d numpy array containing the index positions within an Individual’s non-neutral genotypes object (attribute ‘g’ of an Individual) that correspond to the loci subtending this trait. (This is designed such that an Individual’s genotype for the locus number stored at Trait.loc[i] can be accessed by calling Individual.g[Trait.loc_idx[i], :].) This method of tracking allows Individuals’ full genotypes to be stored in the tskit tables while they carry their non-neutral genotypes with them (a computational optimization).

loci
A numerically sorted 1d numpy array containing the locus numbers of all loci subtending this trait.

lyr_num
Index number of the Landscape layer to which this trait corresponds (i.e. of the environmental variable that serves as the selective force on this trait)

max_alpha_mag
Maximum absolute value that can be drawn for a locus’ effect size (i.e. alpha). Effect sizes are clipped to the closed interval [-max_alpha_mag, max_alpha_mag] when the model is parameterized.

mu
Mutation rate for this trait (expressed in mutations per locus per timestep)

name
The string name of the trait

n_loci
Number of loci subtending this trait
phi
Polygenic selection coefficient (i.e. the selection coefficient acting on the phenotypes, rather than the
genotypes, of this Trait)

univ_adv
Whether or not this trait will be treated as universally advantageous. When False (the default behavior),
individuals will be fitter when their phenotypic values for this trait more closely align with the local
environmental value of the landscape layer corresponding to this trait (i.e. selection will be spatially
varying). When False, phenotypes closer to 1 will be more advantageous globally on the Landscape.

geneomics.structs.genome.read_pickled_genomic_architecture(filename)

geneomics.structs.individual module

Defines the Individual class, with its associated methods and supporting functions

class geneomics.structs.individual.Individual(idx, x, y, age=0, new_genome=None,
                                             sex=None)

Bases: object

Representation of an individual of a given species.

Multiple Individuals are collected as serial integer-keyed values within a Species dict. Those serial indices
continually increment through model time as new Individuals (i.e. offspring) are produced, such that no two
Individuals within the full history of a simulation will ever have the same index number.

NOTE
that correspond to many of the following attributes.

Type For more detail, see the documentation for the parameters

age
The Individual’s age (expressed in timesteps since its birth timestep, when the individual was instantiated
at age 0)

e
The Individual’s current environmental values, organized as a 1d numpy array of length ==
len(Landscape), where the value stored at Individual.e[i] gives the Individual’s current environmental
value for Layer number i (i.e. Landscape[i])

fit
The Individual’s current fitness (calculated as a combination of the Individual’s phenotypes for all traits,
its current environmental values for all corresponding Landscape Layers, and any deleterious loci for
which it has ‘1’ alleles; see documentation for further details)

g
The Individual’s non-neutral genotypes, stored as an L_n x 2 numpy array, where L_n is the current num-
ber of non-neutral loci in the Individual’s Species. Individual’s only carry copies of their non-neutral
genotypes (stored in this attribute). This is a computational optimization, as it allows fitness-based op-
erations to be calculated quickly on the fly (using this attribute’s data), while minimizing the memory
required to stored the full (neutral and non-netural) genotypes of all current Individuals and their an-
cestors (i.e. the ‘spatial pedigree’ stored in the succinct set of tskit tables). The successive rows in
this array store the genotypes corresponding to the loci indicated by the successive locus numbers in
Species.GenomicArchitecture.nonneut_loci.

sex
The Individual’s sex (0=female, 1=male; None, if the Species is unsexed)
The Individual’s current x (i.e. longitudinal) position (in continuous space, bounded within [0, Landscape.dims[0]])

y
The Individual’s current y (i.e. latitudinal) position (in continuous space, bounded within [0, Landscape.dims[1]])

z
The Individual’s phenotypes for each, organized as a 1d numpy array of length == len(Species.traits), where the value stored at Individual.z[i] gives the Individual’s fitness for Trait number i (i.e. Species.traits[i]).

geonomics.structs.landscape module

Defines the Layer and Landscape classes, with their associated methods and supporting functions

class geonomics.structs.landscape.Landscape (lyrs, res=(1, 1), ulc=(0, 0), prj=None, mod=None)

Bases: dict

Representation of a multi-layer (i.e. multivariate) landscape.
Organized as a dict of multiple, serial integer-keyed Layer objects,
Because the Landscape class inherits from dict, Layers can be indexed out using their index-number keys (e.g. mod.land[<idx>]).
The Landscape is stored as the ‘mod.land’ attribute of its corresponding Model object.

NOTE
that correspond to many of the following attributes.

Type For more detail, see the documentation for the parameters
dim
The x,y (i.e. lon,lat; or i,j in array terms) dimensions of the Layer. (Must be the same as the Landscape to which the Layer belongs.)
n_lyrs
The number of Layers in the Landscape

prj
The projection of the Layer (formatted as a PROJ4 string). (Must be the same as the Landscape to which the Layer belongs.)

res
The x,y (i.e. lon,lat; or i,j in array terms) spatial resolution (i.e. cell sizes) of the layer. (Must be the same as the Landscape to which the Layer belongs.)

ulc
The x,y (i.e. lon,lat; or i,j in array terms) coordinates of the upper left corner of the layer. (Must be the same as the Landscape to which the Layer belongs.)

class geonomics.structs.landscape.Layer (rast, lyr_type, name, dim, res=(1, 1), ulc=(0, 0), prj=None, coord_prec=0, units="", scale_min=0, scale_max=1)

Bases: object

Representation of a single environmental layer (i.e. variable).
Multiple Layers are collected as serial integer-keyed values within a Landscape dict.
NOTE

that correspond to many of the following attributes.

**Type**  For more detail, see the documentation for the parameters

**coord_prec**

The precision (i.e. number of significant digits) to which coordinate values should be round when the
Layer is plotted.

**dim**

The x,y (i.e. lon,lat; or i,j in array terms) dimensions of the Layer. (Must be the same as the Landscape to
which the Layer belongs.)

**idx**

Index number of the Layer (i.e. its key within the Landscape dict)

**name**

The string name of the Layer

**prj**

The projection of the Layer (formatted as a PROJ4 string). (Must be the same as the Landscape to which
the Layer belongs.)

**rast**

The 2d numpy array, of shape \texttt{Landscape.dim}, containing the environmental values for this Layer.

**res**

The x,y (i.e. lon,lat; or i,j in array terms) spatial resolution (i.e. cell sizes) of the layer. (Must be the same
as the Landscape to which the Layer belongs.)

- **type:** A string indicating the type of the Layer (‘random’, ‘defined’, ‘file’, or ‘nlmpy’)
- **ulc:** The x,y (i.e. lon,lat; or i,j in array terms) coordinates of the upper left corner of the layer.
  (Must be the same as the Landscape to which the Layer belongs.)
- **units:** A string representation of the units of the Layer’s variable (to be used for plotting).

\texttt{geonomics.structs.landscape.read_pickled_land (filename)}

**geonomics.structs.species module**

Defines the Species class, with its associated methods and supporting functions

\texttt{class geonomics.structs.species.Species (name, idx, inds, land, spp_params, ge-
nomic_architecture=None)}

Representation of the total simulated population of a species.

Organized as a collections.OrderedDict of serial integer-keyed Individuals. Those serial indices continually
increment through model time as new Individuals (i.e. offspring) as produced, such that no two Individuals
within the full history of a simulation will ever have the same index number.

Because the Species class inherits from \texttt{collections.OrderedDict}, Individuals can be indexed out using their
index-number keys (e.g. \texttt{Species[<idx>]}).

All Species within a simulation are stored as serial integer-keyed values within the Community dict (which in
turn is stored at the Model’s ‘mod.comm’ attribute).

NOTE

that correspond to many of the following attributes.

**Type**  For more detail, see the documentation for the parameters that
b
The Species’ intrinsic birth rate, expressed as the probability (in a Bernoulli random draw) that an other-}
wise eligible potential mating pair will successfully reproduce

burned
A bool flag indicating whether or not the Species has been burned in.

cords
A 2d numpy array, length(Species) x 2, composed of columns containing the current x and y locations of all Individuals. Row order is equivalent to the order of individuals in the Species’ OrderedDict.

cells
A 2d numpy array, length(Species) x 2, composed of columns containing the current j and i cell numbers of all Individuals. Row order is equivalent to the order of individuals in the Species’ OrderedDict. (Note the order of the columns! These are not in the i,j order that would be necessary in order use them to subject a Layer.rast object!)

choose_nearest_mate
A bool flag indicating whether or not Individuals should always choose their nearest neighbors as their potential mates

d_max
The maximum probability of death that can be assigned to any Individual

d_min
The minimum probability of death that can be assigned to any Individual

density_grid_window_width
The width of the sliding window that is used by the Species’ _DensityGridStack object when estimating the Species’ population-density array.

direction_distr_[mu/kappa]
The $\mu$ and $\kappa$ parameters of the von Mises distribution from which the direction components of Individuals’ movement vectors will be drawn. (These parameters are only utilized if the Species is parameterized without a _MovementSurface.)

dispersal_distance_distr
A string indicating which distribution to use as the Species’ dispersal-distance distribution (‘lognormal’, ‘wald’, or ‘levy’)

dispersal_distance_distr_param[1/2]
The values of the first and second parameters of the Species’ dispersal-distance distribution (named abstractly like this because they could parameterize a Lognormal, Wald, or Levy distribution, depending on the value of Species.dispersal_distance_distr)

extinct
A bool flag indicating whether or not the Species has gone extinct.

idx
Index number of the Species (i.e. its key within the Community dict)

inverse_dist_mating
A bool flag indicating whether or not the probabilities that an Individual chooses any of its neighbors (i.e. other Individuals within Species.mating_radius distance of them) as their potential mates should vary inversely with the distances between the Individual and each of those neighbors. If False, any neighbor could be chosen with equal probability.

gen_arch
The Species’ genomic architecture (as a GenomicArchitecture object)
The Species’ current carrying-capacity raster. This is a numpy array of shape \textit{Landscape.dim}, with each value indicating the carrying capacity of that cell in the Landscape. It is generated as the product of the Species’ \textit{K\_factor} and the raster array of the Layer provided as the basis of the Species’ carrying capacity (i.e. \textit{spp.K\_factor} * \textit{mod.land[spp.K\_layer].rast}). This array will change during a simulation if the corresponding Layer is parameterized to undergo environmental change.

\textbf{K\_factor}

The factor by which to multiply the Species’ chosen carrying capacity Layer (\textit{mod.land[spp.K\_layer]}) in order to generate the Species’ carrying-capacity raster (\textit{spp.K}).

\textbf{K\_layer}

The index number of the Layer whose raster will be multiplied by \textit{spp.K\_factor} in order to generate the Species’ carrying-capacity raster (\textit{spp.K}).

\textbf{mating\_radius}

The mating radius, expressed in Landscape cell widths, from within which an Individual can choose a potential mate. This will only be used if \textit{Species.choose\_nearest\_mate} is False.

\textbf{max\_age}

The maximum age, in time steps, that any Individual of the Species may reach (after which point they will be culled from the population by imposed mortality)

\textbf{max\_ind\_idx}

The maximum Individual index number that has currently been used

\textbf{move}

A bool flag indicating whether or not Individuals of the Species can move after their initial dispersal as offspring.

\textbf{movement\_distance\_distr}

A string indicating which distribution to use as the Species’ movement-distance distribution (‘lognormal’, ‘wald’, or ‘levy’)

\textbf{movement\_distance\_distr\_param[1/2]}

The values of the first and second parameters of the Species’ movement-distance distribution (named abstractly like this because they could parameterize a Lognormal, Wald, or Levy distribution, depending on the value of \textit{Species.movement\_distance\_distr})

\textbf{mutate}

A bool flag indicating whether or not the Species’ simulation involves mutation.

\textbf{mut\_log}

A bool flag indicating whether or not a mutation log should be written for the Species.

\textbf{N}

The Species’ current population-density raster. This is a numpy array of shape \textit{Landscape.dim}, with each value indicating the population density of that cell in the Landscape, as estimated by the Species’ \_DensityGridStack instance.

\textbf{Nt}

A chronologically ordered list, starting from the first timestep of the burn-in, containing the Species’ total population size at each time step.

\textbf{n\_births}

A chronologically ordered list, starting from the first timestep of the burn-in, containing the Species’ number of births at each time step.

\textbf{n\_births\_distr\_lambda}

If \textit{Species.n\_births\_fixed} is False, then this value serves as the $\lambda$ parameter of the Poisson distri-
bution from which is drawn the number of offspring a mating pair will produce. If `Species.n_births_fixed` is True, then this is the fixed number of offspring that each successful mating pair will produce.

**n_births_fixed**
A bool flag indicating whether or not the number of births should be fixed at `Species.n_births_distr_lambda`

**n_deaths**
A chronologically ordered list, starting from the first timestep of the burn-in, containing the Species’ number of deaths at each time step.

**name**
The string name of the Species

**R**
The intrinsic growth rate of the Species, to be used as ‘R’ in the stochastic, spatialized logistic growth model that controls population dynamics

**repro_age**
The age, in time steps, of first reproduction

**selection**
A bool flag indicating whether or not the Species’ simulation involves selection.

**sex**
A bool flag indicating whether or not this Species is sexed (i.e. whether or not the eligibility of potential mating pairs should be based on whether or not the Individuals in the potential pair are of opposite sexes)

**sex_ratio**
The sex ratio of the Species. This value is expressed as the proportion of all offspring that are males, such that it can be easily used as the probability parameter for the Bernoulli draw of an offspring’s sex. (Note that this value is derived from the ‘sex_ratio’ parameter provided in the Model’s parameters file, but unlike in the parameters file this value is not expressed in as a ratio of males to females.)

**start_N**
The number of Individuals to be generated for the starting population of the Species (i.e. the population size at the beginning of the burn-in)

**t**
The index of the last completed time step run for this Species. This attribute holds the value -1 until the Species has been burned in, at which point it begins incrementing, Pythonically, from 0 (such that a Species with `t == 999` has been run for 1000 time steps).

`geonomics.structs.species.read_pickled_spp(filename)`

---

**Module contents**

`geonomics.utils package`

**Submodules**

`geonomics.utils.io module`

Defines core IO functions
geonomics.utils.spatial module

Defines core spatial classes, with their associated methods and supporting functions

geonomics.utils.viz module

Defines core functions for visualization

Module contents

Submodules

geonomics.main module

Geonomics

Provides: 1. A function for creating custom-structured, editable parameters files. 2. A function for creating model objects from parameters files. 3. Functions for running models. 4. Functions for model visualization. 5. A variety of parameters, for construction of arbitrarily complex simulation scenarios.

How to use the documentation

There are two levels of documentation available: - Basic, procedural documentation is available as docstrings in the code, which are returned when requesting Python’s help on a class or function. - Detailed documentation is available at the Geonomics homepage.

All documentation examples assume that geonomics has been imported as gnx:

```
>>> import geonomics as gnx
```

To get help, call Python’s help function on a component of Geonomics that you are trying to use, such as:

```
>>> help(gnx.make_parameters_file)
```

Or use the question mark, if you are working in the IPython shell or in the Jupyter Notebook:

```
>>> geonomics.make_parameters_file?
```

geonomics.main.make_community (landscape, params)

Make a Community object.

Use the parameter values stored in the input params argument, and the Landscape object input as the landscape argument, to instantiate a Community object.

Parameters

- **landscape** (Landscape) – A Landscape object (such as that returned by the make_landscape function).
- **spp_params** (ParametersDict) – A ParametersDict object (such as that output by the make_params_dict function).

Returns A Community instance.

Return type geonomics.structs.community.Community
geonomics Documentation, Release 1.1.4

geonomics.main.make_genomic_architecture(params, landscape)

Make a GenomicArchitecture object.

Use the parameter values stored in the input params argument, and the Landscape object input as the landscape argument, to instantiate a GenomicArchitecture object.

Parameters

- **params** (ParametersDict) – A ParametersDict object (such as that output by the make_params_dict function).
- **landscape** (Landscape) – A Landscape object (such as that returned by the make_landscape function).

Returns

A GenomicArchitecture instance.

Return type: geonomics.structs.genome.GenomicArchitecture

geonomics.main.make_individual(idx, genomic_architecture=None, new_genome=None, dim=None, parental_midpoint=None, sex=None, age=0)

Make an Individual object.

Make a new Individual, whose index number, genome, location, sex, and age are determined by the input arguments.

Parameters

- **idx** (int) – The index number to be assigned to the new individual.
- **genomic_architecture** (GenomicArchitecture, optional) – A GenomicArchitecture object, describing the genomic characteristics of the Species to which the Individual belongs. If the new_genome argument is not provided then the relevant parameter values in the GenomicArchitecture will be used when randomly drawing the individual’s new genome.
- **new_genome** (nx2 np.ndarray, optional) – The Individual’s genome, as an nx2 Numpy ndarray of 0’s and 1’s.
- **dim** ((int, int), optional) – A tuple of the landscape’s (x, y) dimensionality, to be used if the individual’s location is to be randomly randomly drawn from anywhere on the landscape (rather than being drawn by dispersal from the parental midpoint, which will occur if the parental_midpoint argument is provided).
- **parental_midpoint** ((float, float), optional) – A tuple of the (x,y) continuous coordinates, representing the point from which dispersal should occur to determine the individual’s location. This is normally thought of as the midpoint between the individual’s two parents, but if you are creating an individual without reference to parents then this can also simply be thought of as representing the center of the neighborhood into which the individual will disperse.
- **sex** (int, optional) – An integer representing the individual’s sex (0 for female, 1 for male). Can be left as None for asexual or non-sexed Species.
- **age** (int, optional) – An integer representing the number of timesteps for which the individual has already survived - hence its age, or stage.

Returns

An Individual instance.

Return type: geonomics.structs.individual.Individual

geonomics.main.make_landscape(params)

Make a Landscape object.

Use the parameter values stored in the input params argument to instantiate a Landscape object.
Parameters **params** ([`ParametersDict`]) – A ParametersDict object, such as that output by the `make_params_dict` function.

Returns A Landscape instance.

Return type `geonomics.structs.landscape.Landscape`

```
geonomics.main.make_model(parameters=None, verbose=False)
```

Create a new Model object.

Use either a ParametersDict object or the path to a valid Geonomics parameters file (whichever is provided to the ‘parameters’ argument) to create a new Model object.

**Parameters**

**parameters**

({`ParametersDict`, `dict`, `str`}, optional, default: `None`) – The parameters to be used to make the Model object.

If **parameters** is a ParametersDict object, the object will be used to make the Model.

If **parameters** is a dict object, Geonomics will attempt to convert it to a ParametersDict object, then use that object to make the Model.

If **parameters** is a string, Geonomics will call `gnx.read_parameters_file` to make a ParametersDict object, then use that object to make the Model.

If **parameters** is `None`, or is not provided, then Geonomics will attempt to find a single parameters file in the current working directory with the filename “GNX_params_<...>.py”, will use that file to make a ParametersDict object, then will use that object to make the Model.

Returns An object of the Model class

Return type `geonomics.Model`

Raises

- `ValueError` – If the **parameters** argument was not provided and a single, valid Geonomics parameters file could not be identified in the current working directory
- `ValueError` – If the **parameters** argument was given a string that does not point to a valid parameters file
- `ValueError` – If the ParametersDict provided to the **parameters** argument, or created from the parameters file being used, cannot be successfully made into a Model

See also:

```
`gnx.read_parameters_file()`, `sim.model.Model()`
```

Examples

Make a Model from a single, valid “GNX_params_<...>.py” file that can be found in the current working directory (such as a file that would be produced by calling `gnx.make_parameters_file` without any arguments).

```
>>> gnx.make_model()
<class 'sim.model.Model'>
Model name: GNX_params_13-10-2018_15:54:03
Layers: 0: '0'
Species: 0: '0'
Number of iterations: 1
Number of burn-in timesteps (minimum): 30
Number of main timesteps: 100
Geo-data collected: {}
```

(continues on next page)
Make a Model from a file called `null_model.py`, in the current working directory.

```python
>>> gnx.make_model('null_model.py')
<class 'sim.model.Model'>
```

**Model name**: null_model

**Layers**: 0: 'tmp'
1: 'ppt'

**Species**: 0: 'C. fasciata'

**Number of iterations**: 2500

**Number of burn-in timesteps (minimum)**: 100

**Number of main timesteps**: 1000

**Geo-data collected**: {csv, geotiff}

**Gen-data collected**: {vcf, fasta}

**Stats collected**: {maf, ld, mean_fit, het, Nt}

---

geonomics.main.make_parameters_file

Create a new parameters file.

Write to disk a new, template parameters file. The file will contain the numbers and types of sections indicated by the parameters fed to this function. The new file can then be used ‘out of the box’ to make a new Model object, but typically it will be edited by the user to stipulate the scenario being simulated, then used to instantiate a Model.

**Parameters**

- **filepath** *(str, optional, default: None)* – Where to write the resulting parameters file, in /path/to/filename.py format. Defaults to None. If None, a file named “GNX_params_<datetime>.py” will be written to the working directory.

- **layers** *(int, list of dicts, optional, default: 1)* – Number (and optionally, types) of Layer-parameter sections to include in the parameters file that is generated. Defaults to 1. Valid values and their associated behaviors are:

  **If an int is passed**: Add sections for the stipulated number of Layers, each with default settings:

  - parameters for creating Layers of type ‘random’ (i.e. Layers that will be generated by interpolation from randomly valued random points)
  - no LandscapeChanger parameters

  **If [dict, . . . , dict] is passed**:

  Each dict in this list should have the following key-value pairs:

  KEY | VALUE
  --- | ---
  ‘type’: {‘random’, ‘defined’, ‘file’, ‘nlmpy’}
  ‘change’: bool

  This will add one section of Layer parameters, with the contents indicated, for each dict in this list.

- **species** *(int, list of dicts, optional, default: 1)* – Number (and optionally, types) of Species-parameter sections to include in the parameters file that is generated. Defaults to 1. Valid values and their associated behaviors are:

  **If an int is passed**: Add sections for the stipulated number of Species, each with default settings:
parameters for movement and dispersal without _ConductanceSurfaces
parameters for a GenomicArchitecture with 0 Traits (i.e. with only neutral loci)
no _SpeciesChanger parameters

If [dict, . . . , dict] is passed: Each dict in this list can contain any of the following key-value pairs:

KEY VALUE
'movement': bool,
'movement_surface': bool,
'dispersal_surface': bool,
'genomes': {bool, ‘custom’},
'n_traits': int,
'demographic_change': int,
'parameter_change': bool

This will add one section of Species parameters, customized as indicated, for each dict in the list. (Note that if the ‘genomes’ argument is True or ‘custom’, a section for parameterization of the genomic architecture will be added, but if it is ‘custom’ then a template custom genomic architecture file will also be created (a CSV file), which can be filled in to stipulate the locus-wise values for starting allele frequency, recombination rate, dominance, associated traits, and effect sizes.)

• **data** *(bool, optional, default: False)* — Whether to include a Data-parameter section in the parameters file that is generated. Defaults to None. Valid values and their associated behaviors are:

None, False: Will not add a section for parameterizing data to be collected. No _DataCollector will be created for the Model object made from the resulting parameters file, and no data will be collected during the model runs.

True: Will add a section that can be used to parameterize which data will be collected during the model runs, when, and what file formats will be used to write it to disk. (This which will be managed by the model’s _DataCollector object.)

• **stats** *(bool, optional, default: False)* — Whether to include a Stats-parameter section in the parameters file that is generated. Defaults to None. Valid values and their associated behaviors are:

None, False: Will not add a section for parameterizing the statistics to be calculated. No _StatsCollector will be created for the Model object made from the resulting parameters file, and no statistics will be calculated during the model runs.

True: Will add a section that can be used to parameterize which statistics will be calculated during the model runs, and when. (This will be managed by the model’s _StatsCollector object.)

**Returns** Returns no output. Resulting parameters file will be written to the location and filename indicated (or by default, will be written to a file named “GNX_params_<datetime>.py” in the working directory).

**Return type** None

See also:
sim.params.make_parameters_file()
Notes

All parameters of this function are optional. Calling the function without providing any parameters will always produce the parameters file for the default model scenario. This file can be instantiated as a Model object and run without being edited. Those three steps (create default parameters file; create model from that parameters file; run the model) serve as a base case to test successful package installation, and are wrapped around by the convenience function `gnx.run_default_model`.

Examples

In the simplest example, we can create a parameters file for the default model. Then (assuming it is the only Geonomics parameters file in the current working directory, so that it can be unambiguously identified) we can call the `gnx.make_model` function to create a Model object from that file, and then call the Model.run method to run the model (setting the ‘verbose’ parameter to True, so that we can observe model output).

```python
>>> gnx.make_parameters_file()
>>> mod = gnx.make_model()
```

NOTE: Using the following file, in the current working directory to create the Model object:

GNX_params_21-01-2020_17:22:08.py

```python
>>> mod.run(verbos=True)
```

We could also use some of this function’s arguments, to create a parameters file for a model with 3 Layers and 1 Species (all with the default components for their sections of the parameters file) and with a section for parameterizing data collection.
As a more complex example that is likely to be similar to most use cases, we can create a parameters file for a model scenario with:

- 2 Layers (one being an nlmpy Layer that will not change over model time, the other being a raster read in from a GIS file and being subject to change over model time);
- 2 Species (the first having genomes, 2 Traits, and movement that is dictated by a _ConductanceSurface; the second not having genomes but having dispersal determined by a _ConductanceSurface, and undergoing demographic change)
- data-collection;
- stats-collection;

We can save this to a file named “2-spp_2-trait_model.py” in our current working directory.

```python
>>> gnx.make_parameters_file(layers = 3, data = True)

#list of 2 dicts, each containing the values for each Layer's parameters section
layers = [
    {'type': 'nlmpy'},  #layer 1
    {'type': 'gis', 'change': True},  #layer 2
],

#list of 2 dicts, each containing the values for each Species's parameters section
species = [
    {'genomes': True, 'n_traits': 2, 'movement': True, 'movement_surface': True},  #spp 1
    {'genomes': False, 'movement': True, 'dispersal_surface': True, 'demographic_change': True},  #spp 2
],

#arguments to the data and stats parameters
data = True, stats = True,

#destination to which to write the resulting parameter file
filepath = '2-spp_2-trait_model.py')
```

geonomics.main.make_params_dict(params, model_name=None)

Create a ParametersDict object from a dict object.

Use a plain Python dict object, and an optional model name, to create a new ParametersDict object.

**Parameters**

- `params ((dict))` – The plain Python dict that is to be turned into a Geonomics ParametersDict object.
- `model_name ((str), optional, default: None)` – The name to be assigned to the model. (If not provided, the model will be called ‘unnamed_model’.)

**Returns** A dict of nested dicts, all of which have key-value pairs whose values can be accessed using typical dict notation or using dot notation. If formatted correctly, can be fed into the geonomics.make_model function to create a Geonomics model.

**Return type** geonomics.ParametersDict
Make a Species object.

Use the parameter values stored in the input `spp_params` argument, and the Landscape object input as the `landscape` argument, to instantiate a Species object.

**Parameters**

- `landscape` ([Landscape](#)) - A Landscape object (such as that returned by the `make_landscape` function).
- `spp_params` ([ParametersDict](#)) - The subsection of a full ParametersDict object (such as that output by the `make_params_dict` function) that pertains to the particular species to be created.

**Returns** A Species instance.

**Return type** `geonomics.structs.species.Species`

---

Create a new ParametersDict object.

Read the Geonomics parameters file saved at the location indicated by `filepath`, check its validity (i.e. that all the Layers and Species parameterized in that file have been given distinct names), then use the file to instantiate a ParametersDict object.

**Parameters**

- `filepath` ([str](#)) - String indicating the location of the Geonomics parameters file that should be made into a ParametersDict object.

**Returns** A dict of nested dicts, all of which have key-value pairs whose values can be accessed using typical dict notation or using dot notation. Can be fed into the `geonomics.make_model` function to create a Geonomics model.

**Return type** `geonomics.ParametersDict`

**Raises** `AssertionError` – If either the Layers or the Species parameterized in the parameters file have not all been given distinct names

**See also:**

`sim.params.read()`, `sim.params.ParametersDict()`

**Examples**

Read a parameters file called “null_model.py” (located in the current working directory).

```python
>>> gnx.read_parameters_file('null_model.py')
<class 'sim.params.ParametersDict'>
Model name: null_model
```

---

Run the default Geonomics model.

This will create a parameters file for the default Geonomics model, read that in as a model, and run a single iteration of that model.

**Parameters**

- `delete_params_file` ([bool](#), optional, default: True) – Whether or not to delete the parameters file used for this model after the model has finished running
- `animate` ([bool](#), optional, default: False) – Whether or not plot the model’s species as an animated Matplotlib plot while the model is running.
Returns The Model object that was created, after it has run to completion. (If saved to a variable, it can be used for any number of additional runs using the Model.walk method. It can also be used to try out visualization methods, or to introspect its structure.)

Return type geonomics.Model

geonomics.main.run_demo(name, save_figs=False, time_it=False, **kwargs)

Run a Geonomics demo.

This will run whichever of the Geonomics demos is stipulated by the name argument.

Parameters

• name (str) – Can be one of the following values:
  - 'IBD IBE': Runs the IBD, IBE demo (example 1 from the methods paper).
  - 'simult select': Runs the simultaneous selection demo (example 2 from the methods paper).
  - 'Yosemite': Runs the Yosemite demo (example 3 from the methods paper).

• save_figs (bool, optional, default: False) – If True, all figures that are produced will be saved to their default filenames in the current working directory.

• time_it (bool, optional, default: False) – If True, the run time for the main portion of the model (excluding calculation of data for figures and creation of figures, as much as possible) will be calculated, then displayed after the model has run to completion.

Notes

Some of the available demos come from the original Geonomics methods paper, Hart and Wang 2020, “Geonomics: forward-time, agent-based, spatially explicit, and arbitrarily complex landscape genomic simulations”.

Returns The Model object that was created, after it has run to completion. (If saved to a variable, it can be used for any number of additional runs using the Model.walk method. It can also be used to try out visualization methods, or to introspect its structure.)

Return type geonomics.Model

Module contents

Geonomics - A Python package for running complex landscape genomics simulations

3.11 Attribution

This package was written by Drew Ellison Hart, as part of his PhD work. It is available to freely distribute and modify, with proper attribution, under the MIT License.

Should you use Geonomics for research, education, or other purposes, please cite it as:


Should you have any questions or concerns, please feel free to get in touch at drew.hart<at>berkeley<dot>edu!
3.12 Disclaimer

*Geonomics* claims no affiliation with the philosophy and economic ideology *Georgism*, sometimes referred to as ‘geonomics’. It is a portmanteau of geography and genomics.

We just thought it sounded neat, and found it delightfully confusing.
Indices and tables

- genindex
- modindex
- search
Python Module Index

g
geonomics, 99
geonomics.help, 59
geonomics.main, 91
geonomics.ops, 60
geonomics.ops.change, 60
geonomics.ops.demography, 60
geonomics.ops.mating, 60
geonomics.ops.movement, 60
geonomics.ops.mutation, 60
geonomics.ops.selection, 60
geonomics.sim, 81
geonomics.sim.burnin, 60
geonomics.sim.data, 61
geonomics.sim.model, 61
geonomics.sim.params, 81
geonomics.sim.stats, 81
geonomics.structs, 90
geonomics.structs.community, 81
geonomics.structs.genome, 82
geonomics.structs.individual, 85
geonomics.structs.landscape, 86
geonomics.structs.species, 87
geonomics.utils, 91
geonomics.utils.io, 90
geonomics.utils.spatial, 91
geonomics.utils.viz, 91
Index

A
age (geonomics.structs.individual.Individual attribute), 85
alpha (geonomics.structs.genome.Trait attribute), 84
alpha_distr_mu (geonomics.structs.genome.Trait attribute), 84
alpha_distr_sigma (geonomics.structs.genome.Trait attribute), 84

B
b (geonomics.structs.species.Species attribute), 88
burned (geonomics.structs.community.Community attribute), 82
burned (geonomics.structs.species.Species attribute), 88

calc_stats() (geonomics.sim.model.Model method), 61
cells (geonomics.structs.species.Species attribute), 88
choose_nearest_mate (geonomics.structs.species.Species attribute), 88
Community (class in geonomics.structs.community), 88
coord_prec (geonomics.structs.landscape.Layer attribute), 87
cords (geonomics.structs.species.Species attribute), 88

d

d_max (geonomics.structs.species.Species attribute), 88
d_min (geonomics.structs.species.Species attribute), 88
delep_loc_id (geonomics.structs.genome.GenomicArchitecture attribute), 82
delep_loci (geonomics.structs.genome.GenomicArchitecture attribute), 82
delep_loci_s (geonomics.structs.genome.GenomicArchitecture attribute), 83
density_grid_window_width (geonomics.structs.species.Species attribute), 88
dim (geonomics.structs.landscape.Landscape attribute), 86
dim (geonomics.structs.landscape.Layer attribute), 87
dispersal_distance_distr (geonomics.structs.species.Species attribute), 88
dom (geonomics.structs.species.GenomicArchitecture attribute), 83

e
e (geonomics.structs.individual.Individual attribute), 85
extinct (geonomics.structs.species.Species attribute), 88

F
fit (geonomics.structs.individual.Individual attribute), 85

G
go (geonomics.structs.individual.Individual attribute), 85
gamma (geonomics.structs.genome.Trait attribute), 84
gen_arch (geonomics.structs.species.Species attribute), 88
GenomicArchitecture (class in geonomics.structs.genome), 82
geonomics (module), 99
gonomics.main (module), 59
gonomics.ops (module), 60
geonomics.ops.change (module), 60
geonomics.ops.demography (module), 60
geonomics.ops.mating (module), 60
geonomics.ops.mutation (module), 60
geonomics.ops.selection (module), 60
geonomics.sim (module), 81
geonomics.sim.burnin (module), 60
geonomics.sim.data (module), 61
geonomics.sim.model (module), 61
geonomics.sim.params (module), 81
geonomics.sim.stats (module), 81
geonomics.structs.community (module), 81
geonomics.structs.genome (module), 82
geonomics.structs.individual (module), 85
geonomics.structs.landscape (module), 86
geonomics.structs.species (module), 87
geonomics.utils (module), 91
geonomics.utils.io (module), 90
geonomics.utils.spatial (module), 91
geonomics.utils.viz (module), 91
get_cells() (geonomics.sim.model.Model method), 61
get_coords() (geonomics.sim.model.Model method), 61
get_e() (geonomics.sim.model.Model method), 62
get_fitness() (geonomics.sim.model.Model method), 62
get_genotypes() (geonomics.sim.model.Model method), 62
get_random_individuals() (geonomics.sim.model.Model method), 63
get_tree_sequence() (geonomics.sim.model.Model method), 63
get_x() (geonomics.sim.model.Model method), 63
get_y() (geonomics.sim.model.Model method), 64
get_z() (geonomics.sim.model.Model method), 64

L
L (geonomics.structs.species.GenomicArchitecture attribute), 83
Landscape (class in geonomics.structs.landscape), 86
Layer (class in geonomics.structs.landscape), 86
loc_idx (geonomics.structs.species.Trait attribute), 84
loci (geonomics.structs.species.Trait attribute), 84
lyr_num (geonomics.structs.species.Trait attribute), 84

M
make_community() (in module geonomics.main), 91
make_genomic_architecture() (in module geonomics.main), 92
make_individual() (in module geonomics.main), 92
make_landscape() (in module geonomics.main), 92
make_model() (in module geonomics.main), 93
make_parameters_file() (in module geonomics.main), 94
make_params_dict() (in module geonomics.main), 97
make_species() (in module geonomics.main), 97
mating_radius (geonomics.structs.species.Species attribute), 89
max_age (geonomics.structs.species.Species attribute), 89
max_alpha_mag (geonomics.structs.species.Trait attribute), 89
max_ind_idx (geonomics.structs.species.Species attribute), 89
Model (class in geonomics.sim.model), 61
move (geonomics.structs.species.Species attribute), 89
movement_distance_distr (geonomics.structs.species.Species attribute), 89
mu (geonomics.structs.species.Trait attribute), 84
mu_delet (geonomics.structs.species.GenomicArchitecture attribute), 83
mu_neut (geonomics.structs.species.GenomicArchitecture attribute), 83
mut_log (geonomics.structs.species.Species attribute), 89
mutate (geonomics.structs.species.Species attribute), 89
MutationRateError, 83

N
N (geonomics.structs.species.Species attribute), 89
n_births (geonomics.structs.species.Species attribute), 89
n_births_distr_lambda (geonomics.structs.species.Species attribute), 89
sex_ratio (geonomics.structs.species.Species attribute), 90
SpatialTester (class in geonomics.sim.burnin), 60
Species (class in geonomics.structs.species), 87
start_N (geonomics.structs.species.Species attribute), 90

T
t (geonomics.structs.community.Community attribute), 82
t (geonomics.structs.species.Species attribute), 90
Trait (class in geonomics.structs.genome), 84
traits (geonomics.structs.genome.GenomicArchitecture attribute), 83

U
ulc (geonomics.structs.landscape.Landscape attribute), 86
univ_adv (geonomics.structs.genome.Trait attribute), 85
update() (geonomics.sim.burnin.SpatialTester method), 60
update_counts_cell() (geonomics.sim.burnin.SpatialTester method), 60

W
walk() (geonomics.sim.model.Model method), 78
write_gendata() (geonomics.sim.model.Model method), 80
write_geodata() (geonomics.sim.model.Model method), 80
write_tskit_table_collection() (geonomics.sim.model.Model method), 81

X
x (geonomics.structs.genome.GenomicArchitecture attribute), 83
x (geonomics.structs.individual.Individual attribute), 85

Y
y (geonomics.structs.individual.Individual attribute), 86

Z
z (geonomics.structs.individual.Individual attribute), 86