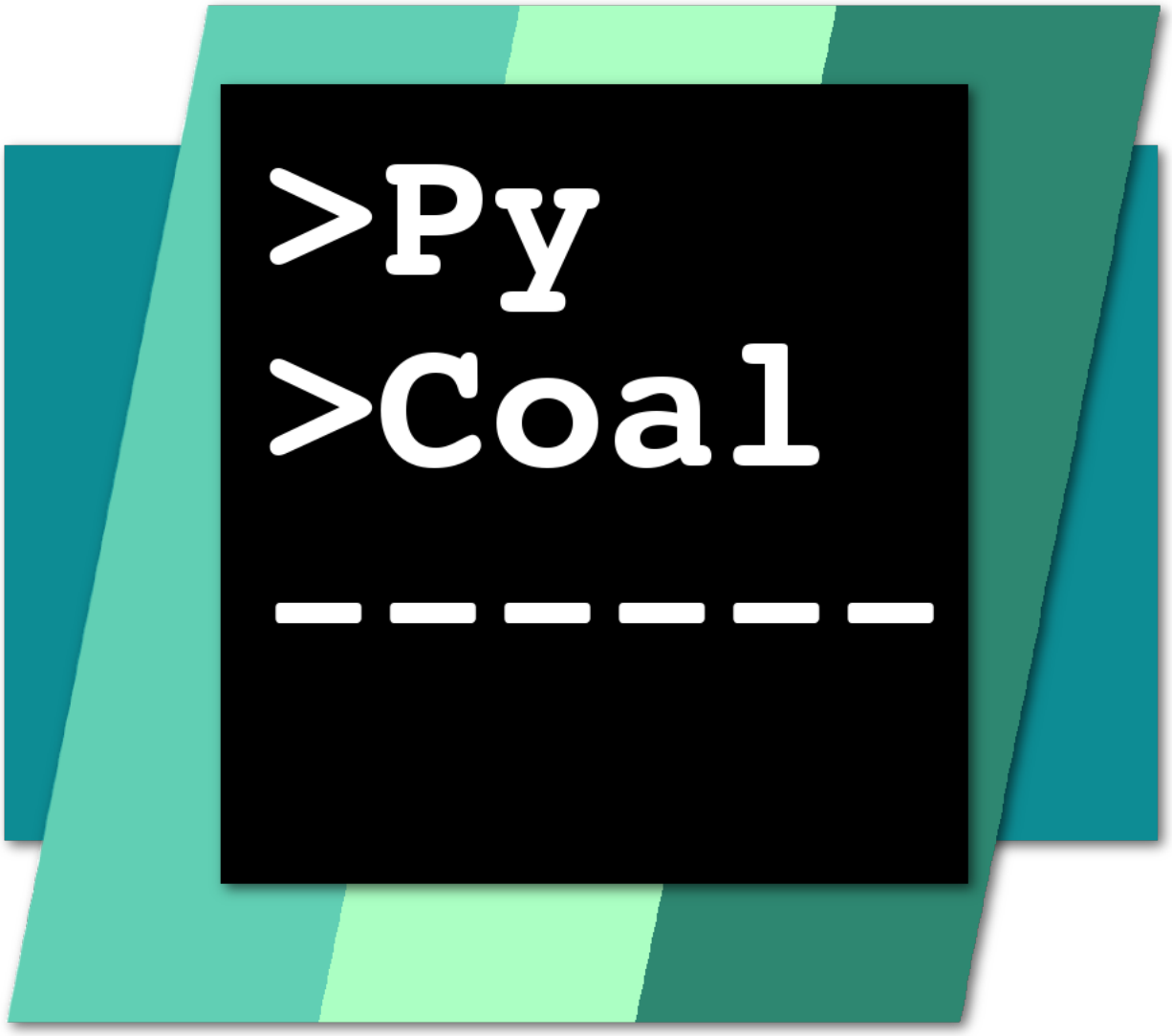


PyCoalescence Documentation

Release 1.1.0rc1

Samuel Thompson

Mar 13, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

PyCoalescence is a python module for the spatially-explicit coalescence neutral simulator, described *here*. In its finished form, PyCoalescence will provide a pythonic interface for setting up, running and analysing spatially-explicit neutral simulations. It allows for much swifter and cleaner creation of configuration files and setting of simulation parameters.

CHAPTER

TWO

HOW-TO

2.1 Installation

Currently, only macOS and linux-based operating systems are supported. Windows compatibility will likely be added at a later date.

2.1.1 Method

Before attempting installation, make sure the *prerequisites are installed*. There are a few options for installation.

- Use in-built installation **[recommended]**

  This is the recommended method for installing the files. Simply run `python setup.py [opts]` where `[opts]` are your required compilation flags (see *Compilation Options*). On some systems this requires that `autoconf` and `autotools` are installed on your computer. The configuration will be calculated,

  > *NECSim* and *SpeciationCounter* will be

  compiled, and the executables will be placed in *build/default/*.

- Use *setup.py* to customise options and install locations. The procedure is the same as exists in *main()*.

```
from PyCoalescence import setup
setup.autoconf() # This may not be necessary and will fail if autotools␣
↪is not installed.
setup.configure(["--with-hpc", "--with-verbose"]) # Replace with desired␣
↪install options
setup.clean()
setup.do_compile()
setup.move_executable("build/custom_install/") # Move to a custom install␣
↪location
```

- Run `./configure` and `make` yourself.

  If you require additional compilation options, run `./configure` with your options from the NECSim/ directory. Then run `make` and check that installation is complete. The executables at NECSim/NECSim and NECSim/SpeciationCounter should then be moved to where they are required.

- Custom compilation

  Compile the c++ files yourself with the required defines and copy the executable to the required directory.

For HPC use, running `python hpc_setup.py` (see *hpc_setup*) will perform compilation for an HPC using the intel compiler and copy the executable to "../../Code" relative to PyCoalescence.

Warning: Additional steps may have to be taken to ensure availability of the correct packages on HPC systems. Check with your HPC administrator for details.

2.1.2 Compilation Options

These are the possible flags which can be provided during installation as options in `python setup.py [opts]`.

Option	Description
–with-debug	Adds additional debugging information
–with-restrict_self	Restricts self-selection during dispersal. Individuals can therefore not become their own parent
–with-verbose	Prints information to the terminal. Otherwise, prints to a log file in *Logs/*
–with-normal_dispersal	Choose a normal distribution for dispersal distances
–with-fat_tail_dispersal	Choose a fat-tailed kernel for dispersal distances
–with-infinite_landscape	Map boundaries are ignored and individuals can disperse to anywhere.
–with-gdal=DIR	Define a gdal library at DIR
–with-hpc	Compile ready for HPC, using intel's icpc compilation and a variety of optimisation flags.
–with-boost=DIR	Define a boost library at DIR

Warning: It is recommended that `--with-verbose` is always used for any HPC system, with the `--with-hpc` flag, as writing to a log file can cause complications in a virtualised system where stdin and stdout calls are redirected already. Compilation will likely fail due to libraries that are not present in the standard library of most HPC systems.

2.2 Performing simulations

2.2.1 Setting up simulations

It is recommended that PyCoalescence is used for simple simulation set up and running. There are two methods for providing configuration options in PyCoalescence. Both require the same initial procedure.

1. Specify simulation parameters

- run *setup()* to link to the correct executables. If you have installed via `python setup.py`, the arguments can be left blank. Otherwise, specify the path to the executables.
- set simulation parameters using *set_simulation_params()* to set the job number, task number, output directory and other key simulation variables.
- set the map variables by one of the following:
  (a) *set_map_parameters()* to input file paths and dimensions
  (b) *set_map_files()* to set the map file paths. Then run *detect_map_dimensions()* to automatically detect file dimensions.

Tip: Check *Limitations of simulation variables* for important information on restrictions on simulation inputs.

Note: *detect_map_dimensions()* requires that the files are in **.tif** formats so that file dimensions can be read. If input files are csv format, method a) should be used.

- Optionally, also run *set_speciation_rates()* to set a list of speciation rates to apply at the end of the simulation.

2. [Optionally] Add map configuration file If you require multiple map files at different points in time, you shall need to create a configuration (.txt or .cfg) file to make these options accessible to the program.
   - First add the pristine map options using *add_pristine_map()* This can be performed multiple times to add several maps.
   - Create the map configuration file (*create_map_config()*)
3. [Optionally] Add temporal sampling configuration file If you require sampling at points other than the present day, these can be specified in another configuration file.
   - Add temporal sampling points using *add_sample_time()* Multiple sample points can be added.
   - Create the temporal sampling config file (*create_temporal_sampling_config()*)
4. [Optionally] Generate the main config file Run *create_config()* to generate the main config file.

Note: If you wish to use multiple map files or multiple temporal samples and wish to use a main config file as well, you must generate call *create_config()* **after** both *create_map_config()* and *create_temporal_sampling_config()*

Warning: It is possible to use temporal config files and map config files without using a main config file. However, if you use a main config file, you **must** use a map config and temporal config.

5. Check and run simulations
   - Run *complete_setup()* to generate the command to be passed to the c++ program.
   - Run *run_checks()* to check that all prerequisites have been fulfilled.
   - Finally, start the simulation using *run_coalescence()*

Note: See *Glossary* for definitions of *sample grid*, *fine map* and *coarse map*.

2.2.2 Examples

A simple simulation

```
from PyCoalescence import Coalescence
# use default executables
c.setup()
# set the main simulation parameters
c.set_simulation_params(job_num=1, job_type=1, output_directory="output", min_
↪speciation_rate=0.1,
                        zfat=4, dispersal=4, deme=1, sample_size=0.1, max_time=1,␣
↪lambda_value=1,
                        min_num_species=1, forest_change_param=0, pristine_forest=200,
↪ time_config_file="null")
# set the speciation rates
c.set_speciation_rates([0.1, 0.2, 0.3])
# set the map parameters - null means the map will be generated with 100% cover␣
↪everywhere (no file input).
c.set_map_parameters(sample_file = "null", sample_x = 100, sample_y=100,
                     fine_file = "null", fine_x = 200, fine_y = 200, fine_x_offset =␣
↪50, fine_y_offset = 50,
                     coarse_file = "null", coarse_x = 1000, coarse_y = 1000,
                     coarse_x_offset = 100, coarse_y_offset = 100, coarse_scale = 10,
                     pristine_fine_map = "null", pristine_coarse_map = "null")
# complete setup and run simulation
c.complete_setup()
c.run_checks()
c.run_coalescence()
```

A more complex example using config files, multiple temporal sampling points and detection of map dimensions from the inputted map files.

```
from PyCoalescence import Coalescence
c = Coalescence()
# use default executables
c.setup()
# set the main simulation parameters
c.set_simulation_params(job_num=1, job_type=1, output_directory="output", min_
↪speciation_rate=0.1,
                        zfat=4, dispersal=4, deme=1, sample_size=0.1, max_time=1,␣
↪lambda_value=1,
                        min_num_species=1, forest_change_param=0, pristine_forest=200,
↪ time_config_file="null")
# set the speciation rates
c.set_speciation_rates([0.1, 0.2, 0.3])
# set the map files
c.set_map_files(sample_file="null", fine_file="path/to/fine.tif", coarse_file="path/
↪to/coarse.tif")
# detect the dimensions of the .tif files
c.detect_map_dimensions()
# add sample times
c.add_sample_time(0.0)
c.add_sample_time(1.0)
```

```
# add pristine maps
c.add_pristine_map(fine_map="path/to/pristinefine1.tif", coarse_map="path/to/
↪pristinecoarse1.tif", time=1, rate=0.5)
# create configuration files
c.create_temporal_sampling_config("timeconf.txt")
c.create_map_config(mapconf.txt)
c.create_config(mainconf.txt)
# complete setup and run checks
c.complete_setup()
c.run_checks()
# run the simulation
c.run_coalescence()
```

Note: NECSim can also be run directly using command line arguments (see *Introduction to NECSim*).

2.2.3 Limitations of simulation variables

Important: This section contains key information about the simulation inputs. Please read carefully to minimise any unnecessary bugs.

Certain simulation variables have limitations, depending on the method of setting up the simulation.

- Map variables set up using *set_map_parameters()*
  - Sample map dimensions must be smaller than fine map dimensions.
  - Fine map dimensions must be smaller than coarse map dimensions (supplied at the resolution of the fine map files).
  - Dimensions of pristine fine and coarse maps must match their respective current map dimensions.
  - All offsets must maintain the smaller map within the larger map
  - If any files are supplied as 'null', map sizes must still be provided. This is important for sample map size, but should be corrected in a future update for coarse map files.

  Todo

  Allow potential for specifying either "null" or "none" map types ("null" creates a map at the specified size, whereas "none" creates hard boundaries without any in-memory map object created at all.

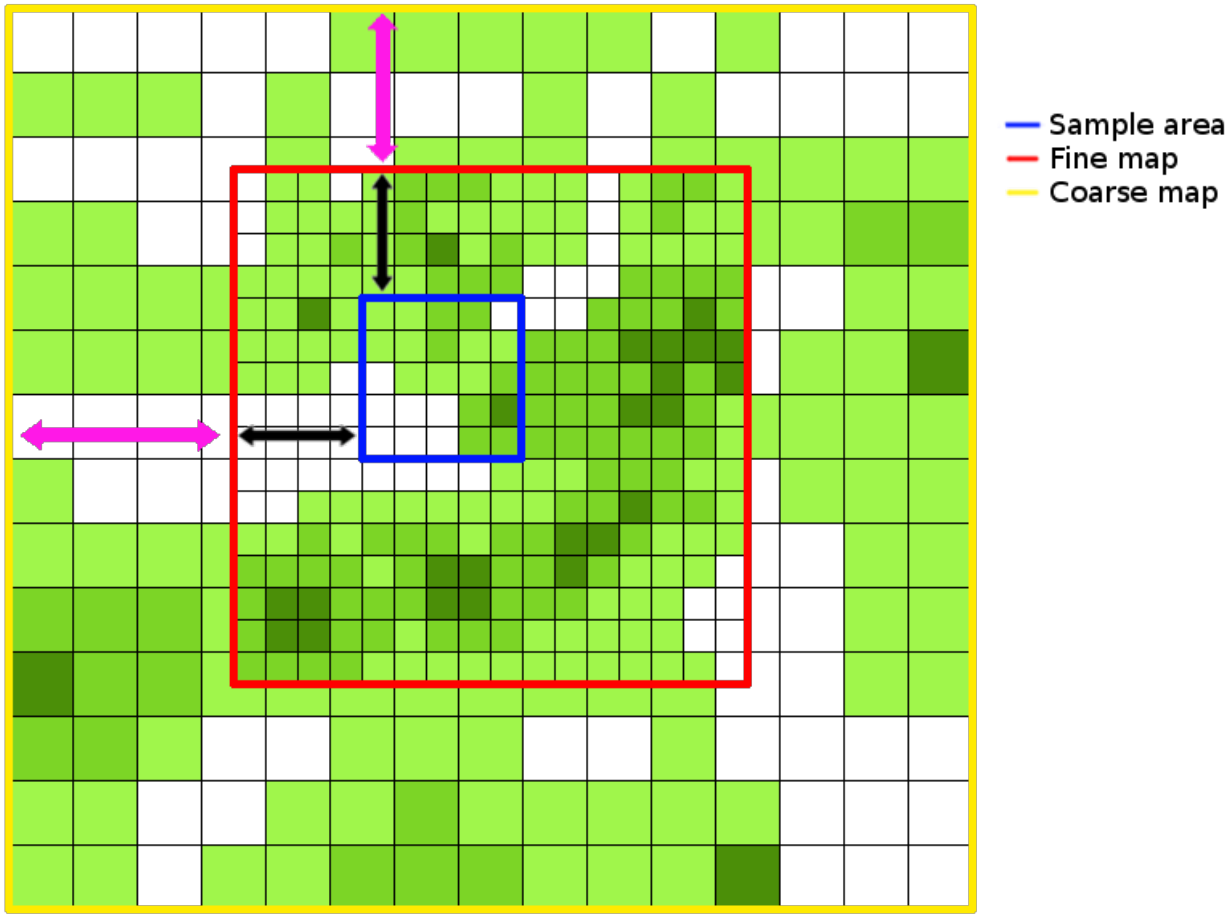
- Map files (and variables) set using *set_map_files()*
  - In addition to the above conditions being true, the files must all be georeferenced, so that coarse and fine map dimensions will be read correctly.

  Hint: Use a GIS program (such as ArcGIS or QGIS) for manipulation of map files to ensure georeferencing is preserved.

- If the samplemask map is "null", the program will read the dimensions from the fine map and choose that as the area to sample entirely over. Supplying "null" will therefore sample the entirety of the fine map.

Hint: Scalings and offsets between maps should also work correctly, but if problems are encountered, try manually specifying offsets and dimensions to identify any problems.

An example of how the map files are related is shown below. Black arrows indicate the offsets for the fine map (in the x and y dimensions) and purple arrows indicate the offsets for the coarse map.



2.3 Post-simulation analysis

Once simulations are complete, the *SpeciationCounter* program can be used to apply additional speciation rates to the coalescence tree. A simple way of applying additional simulation rates is provided within the `Coalescence class`.

The two functions for this routine are

- `set_speciation_params()` which takes as arguments
  - the SQL database file containing a finished simulation
  - T/F of recording full spatial data

  - either a csv file containing fragment data, or T/F for whether fragments should be calculated from squares of continuous habitat. * list of speciation rates to apply
  - [optional] a sample file to specify certain cells to sample from
  - [optional] a config file containing the temporal sampling points desired.
- *apply_speciation()* performs the analysis. This can be extremely RAM and time-intensive for large simulations. The calculations will be stored in extra tables within the same SQL file as originally specified.

The procedure for applying additional speciation rates to an existing database is

```
from PyCoalescence import Coalescence
c = Coalescence()
speciation_rates = [0.1, 0.2 ,0.3]
c.set_speciation_params("output/SQL_data/data_1_1.db", "T", "null", speciation_rates)
c.apply_speciation()
```

2.4 Extended analysis

The *coal_analyse* module can be used for more extensive simulation analysis, such as comparing simulated landscapes to real data, generating species abundance distributions and calculating goodness of fits.

The general procedure for using this module involves a few functions, all contained in the *Tree class*.

- *set_database()* generates the link to the SQL database, which should be an output from a NECSim simulation (probably ran using the *Coalescence class*.
- `import_comparison_data()` reads an SQL database which contains real data to compare to the simulation output. The comparison data should contain the following tables:
  - BIODIVERSITY_ METRICS, containing *only* "metric", "fragment", "value" and "number_of_individuals" columns. The metric can be "fragment_richness" or any other metric created by your own functions which will exists also in the simulated data.
  - FRAGMENT_ABUNDANCES containing *at least* "plot

2.5 Testing install

The system install can be tested by running `test_install.py` from the command line (`python test_install.py`) which will compile the program for a variety of simulation set-ups (first test run only) and run a test suite. Compilation may take some time if it hasn't been run before.

CHAPTER

THREE

PREREQUISITES

3.1 Essential

- The SQLite library available here. Require both `c++` and `python` installations.
- The Boost library for C++ available here.
- C++ compiler (such as GNU g++) with C++11 support.
- Numerical python (`numpy`) package.

3.2 Recommended

- The gdal library for both python and C++ (available here). This is **ESSENTIAL** if you wish to use .tif files for *NECSim*. It allows reading parameter information from .tif files (using *`detect_map_dimensions()`*). Both the python package and `c++` binaries are required; installation differs between systems, so view the gdal documentation for more help installing gdal properly.
- The fast-cpp-csv-parser by Ben Strasser, available here. This provides much faster csv read and write capabilities and is probably essential for larger-scale simulations, but not necessary if your simulations are small. The folder *fast-cpp-csv-parser/* should be in the same directory as your C++ header files (the NECSim directory).

Note: Running `configure` (or `python setup.py`) will detect system components, including `sqlite3`, `boost`, `gdal` and `fast-cpp-csv-parser` and set the correct compilation flags.

CHAPTER

FOUR

GLOSSARY

sample grid The map file containing the points to be sampled from. This is the most zoomed in level of the simulation, and should encompass all starting locations of lineages.

fine map The map file at a higher resolution (the same as the sample grid) which covers the area lineages are most likely to move to, or where a higher spatial resolution is considered important.

coarse map The map file at a lower resolution (specified by the scale) which covers a larger area than the fine map. This is to allow lineages to move far from their starting positions if required.

pristine map A historic map containing population densities at those times. Both fine and coarse pristine maps can be declared and multiple sets of pristine maps can be declared at unique times.

CHAPTER

FIVE

CONTACTS

Author: Samuel Thompson

Contact: samuelthompson14@imperial.ac.uk - thompsonsed@gmail.com

Institution: Imperial College London and National University of Singapore

CHAPTER

SIX

CODE DOCUMENTATION

6.1 Modules

6.1.1 PyCoalescence package

PyCoalescence Package provides the facilities for running spatially-explicit neutral coalescence ecological simulations and performing basic analysis of the simulation outputs. The program requires the NECSim program to function properly, which will be included in the package at a later date.

Module Contents

Submodules

batch_process module

Contains the operations for performing batch analysis on multiple outputs of a PyCoalescence simulation.

coal_analyse module

Contains the operations for performing basic analysis on the output of a PyCoalescence simulation.

class `Tree`

Contains the coalescence tree and performs various calculations of different biodiversity metrics, which are then stored in the SQLite database.

The general process is

- Import the database (set_database()) and import the comparison data, if required (import_comparison_data())
- Calculate required metrics (such as calculate_fragment_richness())
- Save the required metrics to the SQL database (such as save_fragment_richness())
- Optionally, calculate the goodness of fit (calculate_goodness_of_fit())

`calculate_BCI_similarity()`

Calculates the similarity between each fragment and the BCI dataset.

This function is only appropriate for Panama tree neutral simulations based on Center for Tropical Forest Science (CTFS) tree census data

If the fragment species richness has not yet been calculated, calculate_fragment_richness() will be called.

calculate_comparison_octaves(*store=False*)
Calculates the octave classes for the comparison data and for fragments (if required). If the octaves exist in the FRAGMENT_OCTAVES table in the comparison database, the data will be imported instead of being re-calculated. Stores the new octave classes in self.comparison_octaves.

Parameters **store** – if True, stores within the comparison database.

calculate_fragment_octaves()
Calculates the octave classes for each fragment. Outputs the calculated richness into the SQL database within a FRAGMENT_OCTAVES table

calculate_fragment_richness()
Calculates the fragment richness and stores it in a new table. Each time point will be recorded as a new variable.

calculate_goodness_of_fit()
Calculates the goodness-of-fit measure based on the calculated biodiversity metrics, scaling each metric by the number of individuals involved in the metric.

This requires that import_comparison_data() has already been successfully run.

Note that this doesn't calculate anything for values which have not yet been written to the SQL database - make sure that the relevant save functions have been run already.

The resulting value will then be written to the BIODIVERSITY_METRICS table in the SQL database.

calculate_landscape_octaves(*maxval*)
Calculates the octave classes for the landscape. Outputs the calculated richness into the SQL database within a FRAGMENT_OCTAVES table. Called automatically in calculate_fragment_octaves()

Parameters **maxval** – the maximum reference currently existing in the database (for correct outputting).

calculate_landscape_richness()
Calculates the landscape richness from across all fragments and stores result in a new table in SPECIES_RICHNESS Stores a separate result for each speciation rate and time.

calculate_octaves_error()
Calculates the error in octaves classes between the simulated data and the comparison data. Stores each error value as a new entry in BIODIVERSITY_METRICS under fragment_octaves. Calculates the error by comparing each octave class and summing the relative difference. Octaves are then averaged for each fragment.

check_biodiversity_table_exists()
Checks whether the biodiversity table exists and creates the table if required.

Returns the max reference value currently existing

clear_calculations()
Removes the BIODIVERSITY_METRICS and FRAGMENT_OCTAVES tables completely. Note that this cannot be undone (other than re-running the calculations).

dispersal_parameters()
Reads the dispersal parameters from the database and returns them.

Returns a list of the dispersal parameters [zfat, L-value]

fix_double_fragments_abundances()
Fixes the double fragments from cocoli and sherman for the CTFS Panama dataset, combining the two fragment abundances into one biodiversity metric.

fix_double_fragments_octaves()
Fixes the double fragments for cocoli and sherman for the CTFS Panama dataset, combining octave classes for the two sites. Writes to the FRAGMENT_OCTAVES table, removing old entries.

get_fragment_octaves(*fragment*, *speciation_rate*, *time=0.0*)
Get the pre-calculated octave data for the specified fragment, speciation rate and time. This requires self.calculate_fragment_octaves() to have been run successfully at some point previously.

Parameters

- **fragment** – the desired fragment
- **speciation_rate** – the desired speciation rate
- **time** – the desired generation time (defaults to 0.0)

Returns output from FRAGMENT_OCTAVES for the selected variables

get_goodness_of_fit()
Returns the goodness of fit from the file.

Returns the full output from the SQL query

Return type list

get_goodness_of_fit_fragment_octaves()
Returns the goodness of fit for fragment octaves from the file.

Note that no error will be thrown if the table doesn't exist, just an empty list returned. This improves compatibility of functions that call this one.

Returns the full output from the SQL query

Return type list

get_goodness_of_fit_fragment_richness()
Returns the goodness of fit for fragment richness from the file.

Note that no error will be thrown if the table doesn't exist, just an empty list returned. This improves compatibility of functions that call this one.

Returns the full output from the SQL query

Return type list

get_goodness_of_fit_fragment_similarity_bci()
Returns the goodness of fit for fragment similarity with BCI from the file. This is specifically for use in Panama tropical rainforest simulations parameterised by CTFS data.

Note that no error will be thrown if the table doesn't exist, just an empty list returned. This improves compatibility of functions that call this one.

Returns the full output from the SQL query

Return type list

get_job()
Gets the job number (the seed) and the job type (task number).

Returns list containing [job_number (seed), job_type (task number)]

get_landscape_richness(*speciation_rate=None*, *time=None*)
Reads the landscape richness from the SPECIES_RICHNESS table in the database. Returns the richness for each speciation rate and time.

Note that this should produce the same result as get_richness(sr, t) with the corresponding sr and t.

Note that the return type of this function changes based on whether speciation rates and times were supplied. If they were, returns a single integer. Otherwise, returns a list of all species richnesses.

Parameters

- **speciation_rate** – the required speciation rate (optional)
- **time** – the required time

Returns either a list containing the speciation_rate, time, richness OR (if specific speciation rate and time provided), the species richness at that time and speciation rate.

Return type int, list

get_richness(*speciation_rate=None*, *time=None*)

Get the system richness for the supplied speciation rate and time. Note that a richness of 0 is returned if there has been some problem; it is assumed that species richness will be above 0 for any simulation.

Note that the values generated by this method should be identical to those produced by self.get_landscape_richness()

Parameters

- **speciation_rate** – the required speciation rate
- **time** – the required time

Returns the system species richness

get_simulation_parameters()

Reads the simulation parameters from the database and returns them. :return: a list of the simulation parameters [seed, job_type, output_dir, spec_rate, zfat, L_value, deme, sample_size, maxtime, lambda, min_spec, forest_change, time_since_pristine, time_config, coarse_map vars, fine map vars, sample_file, gridx, gridy, pristine coarse map, pristine fine map]

get_species_list()

Gets the entirety of the SPECIES_LIST table, returning a tuple with an entry for each row. This can be used to construct custom analyses of the coalescence tree.

Note: The species list will be produced in an unprocessed format, with

---

Returns a list of each coalescence and speciation event, with locations, performed in the simulation

Return type tuple

import_comparison_data(*filename*)

Imports the SQL database that contains the biodiversity metrics that we want to compare against.

This can either be real data (for comparing simulated data) or other simulated data (for comparing between models).

If the SQL database does not contain the relevant biodiversity metrics, they will be calculated (if possible) or skipped.

The expected form of the database is the same as the BIODIVERSITY_METRICS table, except without any speciation rates or time references, and a new column containing the number of individuals involved in each metric.

Parameters **filename** (*str*) – the file containing the comparison biodiversity metrics.

is_completed()
Indicates whether the simulation has been performed to completion, or if the simulation has been paused and needs to be completed before analysis can be performed. :return: bool: true if simulation is complete

save_fragment_richness()
Saves the fragment richness data, if it has been calculated (otherwise calculates it first) to a new table in the SQL file.

save_fragment_similarity_bci()
Saves the fragment similarity to BCI, if it has been calculated (otherwise calculates it first) to a new table in the SQL file.

set_database(*filename*)
Sets the database to the specified file and opens the sqlite connection.

This must be done before any other operations can be performed and the file must exist.

Note that this will throw an IOError if the simulation is not complete, as analysis can only be performed on complete simulations. However, the database WILL be set before the error is thrown, allowing for analysis of incomplete simulations if the error is handled correctly.

Parameters **filename** – the SQLite database file to import

wipe_data()
Wipes all calculated data apart from the original, unformatted coalescence tree. The Speciation_Counter program will have to be re-run to perform any analyses.

collate_bci_sim(*file_dir*, *filename='Collated_fits.db'*)
Collates the goodness of fit similarity with BCI values from every file in the specified directory and places them in one new file. Note that files with 'collated' in the name will be ignored.

This is specifically for use in Panama tropical rainforest simulations parameterised by CTFS data.

Parameters

- **file_dir** – the file directory to examine
- **filename** – [optional] the output file name.

collate_fits(*file_dir*, *filename='Collated_fits.db'*)
Collates the goodness of fit values from every file in the specified directory and places them in one new file. Note that files with 'collated' in the name will be ignored. Note that if the output file exists, it will be deleted.

Creates three separate tables in the output file, one for overall goodness of fit, one for fragment richness fits, and one for fragment octaves fits. :param file_dir: the file directory to examine :param filename: [optional] the output file name.

coalescence module

Contains the Coalescence and Map classes as part of the PyCoalescence Project.

Operations involve setting up and running simulations, plus basic tree generation after simulations have been completed.

class Coalescence(*coalescence_simulator=None*, *speciation_simulator=None*, *logging_level=20*, *log_output=None*, ***kwargs*)
A class containing the coalescence running routines and methods for setting up the simulation of a variety of different systems

add_pristine_map(*fine_map*, *coarse_map*, *time*, *rate*)
Adds an extra map to the list of pristine maps.

Parameters

- **fine_map** – the pristine fine map file to add
- **coarse_map** – the pristine coarse map file to add
- **time** – the time to add (when the map is accurate)
- **rate** – the rate to add (the rate of forest change at this time)

add_sample_time(*time*)
Adds an extra sample time to the list of times.

This allows for multiple temporal sample points from within the same simulation.

Parameters **time** – the sample time to add

apply_speciation()
Creates the list of speciation options and performs the speciation analysis by calling Speciation_Counter. This must be run after the main coalescence simulations are complete. It will create additional fields and tables in the SQLite database which contains the requested data.

bci_tif_option()
Utilise the tif files instead of the csv files. Provided for increased speed of file parsing.

check_simulation_params()
Checks that simulation parameters have been correctly set and the program is ready for running. Note that these checks have not been fully tested and are probably uneccessary in a large number of cases.

complete_setup()
Completes the setup process by creating the list that will be passed to the c++ executable

create_config(*output_file*)
Generates the output config files. This version creates the concise version of the config file.

Parameters **output_file** (*str*) – the file to generate the config option. Must be a path to a .txt file.

create_map_config(*output_file*)
Generates the map config file from reading the spatial structure of each of the provided files.

Parameters **output_file** (*str*) – the file to output configuration data to (the map config file)

create_temporal_sampling_config(*config_file*)

detect_map_dimensions()
Detects all the map dimensions for the provided files (where possible) and sets the respective values. This is intended to be run after set_map_files() :return:

get_richness(*speciation_rate=None*, *time=None*)
Calls coal_analyse.get_richness() with the supplied variables.

Requires successful import of coal_analyse and sqlite3.

Parameters

- **speciation_rate** – the speciation rate to extract system richness from.
- **time** – the time to extract system richness from

Returns the species richness.

load_config(*config_file*)
Loads the config file by reading the lines in order.

Parameters **config_file** (*str*) – the config file to read in.

panama_tif_option()
Utilise the tif files instead of the csv files for fasting file reading.

resume_sim(*directory*, *job_num*, *job_type*, *max_time*)
Resumes the simulation from the specified directory, looking for the simulation with the specified seed and task referencing.

Parameters

- **directory** – the directory to search for the paused simulation
- **job_num** – the seed of the paused simulation
- **job_type** – the task of the paused simulation
- **max_time** – the maximum time to run simulations for

Returns None

run_checks()
Check that the simulation is correctly set up and that all the required files exist.

run_coalescence()
Attempt to run the simulation with the given simulation set-up. This is the main routine performing the actual simulation which will take a considerable amount of time.

run_simple(*seed*, *task*, *output*, *alpha*, *sigma*, *size*)
Runs a simple coalescence simulation on a square infinite landscape with the provided parameters. This requires a separate compilation of the inf_land version of the coalescence simulator.

Note that this function returns richness=0 for failure to read from the file. It is assumed that there will be at least one species in the simulation.

Parameters

- **seed** – the simulation seed
- **task** – the task (for file naming)
- **output** – the output directory
- **alpha** – the speciation rate (also nu)
- **sigma** – the normal distribution sigma value for dispersal
- **size** – the size of the world (so there will be size^2 individuals simulated)

Returns the species richness in the simulation

set_bci(*job_num*, *job_type*, *zfat*, *dispersal*, *output_directory=None*, *data_directory='./'*)
Sets up the simulation for the BCI and other CTFS plots in Panama (sampling only the CTFS plots). This requires the compute node to have been specified beforehand.

Currently runs for 40 hours then stops the simulation.

set_compute_node(*compute_node*)
Sets the compute node and (for pre-defined nodes) runs the setup routine.

Parameters **compute_node** (*str*) – the reference for the desired compute node, chosen from pre-specified options

set_map_config(*file*)
Sets a specific map config and tells the program that full commmand-line parsing is not required.

Parameters **file** (*str*) – the file to read map config options from

set_map_files(*sample_file*, *fine_file=None*, *coarse_file=None*, *pristine_fine_file=None*, *pristine_coarse_file=None*)

Sets the map files (or to null, if none specified). It then calls detect_map_dimensions() to correctly read in the specified dimensions. Note that if sample_file is "null", values will remain at 0. If coarse_file is "null", it will default to the size of fine_file with zero offset.

Parameters

- **sample_file** – the sample map file. Provide "null" if on samplemask is required.
- **fine_file** – the fine map file. Defaults to "null" if none provided.
- **coarse_file** – the coarse map file. Defaults to "null" if none provided.
- **pristine_fine_file** – the pristine fine map file. Defaults to "null" if none provided.
- **pristine_coarse_file** – the pristine coarse map file. Defaults to "null" if none provided.

:rtype None

:return None

set_map_parameters(*sample_file*, *sample_x*, *sample_y*, *fine_file*, *fine_x*, *fine_y*, *fine_x_offset*, *fine_y_offset*, *coarse_file*, *coarse_x*, *coarse_y*, *coarse_x_offset*, *coarse_y_offset*, *coarse_scale*, *pristine_fine_map*, *pristine_coarse_map*)

Set up the map objects with the required parameters. This is required for csv file usage.

Note that this function is not recommended for tif file usage, as it is much simpler to call set_map_files() and which should automatically calculate map offsets, scaling and dimensions.

Parameters

- **sample_file** – the sample file to use, which should contain a boolean mask of where to sample
- **sample_x** – the x dimension of the sample file
- **sample_y** – the y dimension of the sample file
- **fine_file** – the fine map file to use (must be equal to or larger than the sample file)
- **fine_x** – the x dimension of the fine map file
- **fine_y** – the y dimension of the fine map file
- **fine_x_offset** – the x offset of the fine map file
- **fine_y_offset** – the y offset of the fine map file
- **coarse_file** – the coarse map file to use (must be equal to or larger than fine map file)
- **coarse_x** – the x dimension of the coarse map file
- **coarse_y** – the y dimension of the coarse map file
- **coarse_x_offset** – the x offset of the coarse map file at the resolution of the fine map
- **coarse_y_offset** – the y offset of the coarse map file at the resoultion of the fine map
- **coarse_scale** – the relative scale of the coarse map compared to the fine map (must match x and y scaling)
- **pristine_fine_map** – the pristine fine map file to use (must have dimensions equal to fine map)

- **pristine_coarse_map** – the pristine coarse map file to use (must have dimensions equal to coarse map)

set_panama(*job_num*, *job_type*, *zfat*, *dispersal*, *output_directory=None*, *data_directory=None*)
Sets up the simulation for the whole of Panama. This simulation require a massive amount of RAM and CPU time, so it is advisable to call this function only on HPC clusters.

set_simulation_params(*job_num*, *job_type*, *output_directory*, *min_speciation_rate*, *zfat*, *dispersal*, *deme*, *sample_size*, *max_time*, *lambda_value*, *min_num_species*, *forest_change_param*, *pristine_forest*, *time_config_file*)
Set all the simulation parameters apart from the map objects.

Parameters

- **job_num** (*int*) – the unique job number for this simulation set
- **job_type** (*int*) – the job type (used for easy file identification after simulations are complete)
- **output_directory** (*str*) – the output directory to store the SQL database
- **min_speciation_rate** (*float*) – the minimum speciation rate to simulate
- **zfat** (*float*) – the dispersal zfat value, also called eta.
- **dispersal** (*float*) – the dispersal L value.
- **deme** (*int*) – the deme size (in individuals per cell)
- **sample_size** (*float*) – the sample size of the deme (decimal 0-1)
- **max_time** (*float*) – the maximum allowed simulation time (in seconds)
- **lambda_value** (*float*) – the relative cost of travelling through non-habitat (default is 1)
- **min_num_species** (*int*) – the minimum number of species known to exist
- **forest_change_param** (*float*) – the rate of forest change over time
- **pristine_forest** (*float*) – the time in generations since a pristine state was achieved
- **time_config_file** (*str*) – the path to the time config file (or null)

set_speciation_params(*file*, *record_spatial*, *record_fragments*, *speciation_rates*, *sample_file=None*, *time_config_file=None*)
Set the parameters for the application of speciation rates. If no config files or time_config files are provided, they will be taken from the main coalescence simulation.

Parameters

- **file** (*str*) – the file to read speciation parameters from
- **str record_spatial** (*bool,*) – a boolean of whether to record spatial data
- **record_fragments** (*str*) – either a csv file containing fragment data, or T/F for whether fragments should be calculated from squares of continuous habitat.
- **speciation_rates** (*list*) – a list of speciation rates to apply
- **sample_file** (*str*) – a sample tif or csv specifying the sampling mask
- **time_config_file** (*str*) – a configuration file of temporal sampling points

set_speciation_rates(*speciation_rates*)
Add speciation rates for analysis at the end of the simulation. This is optional

Parameters **speciation_rates** (*list*) – a list of speciation rates to apply at the end of the simulation

setup (*coalescence_simulator='build/default/./NECSim'*, *speciation_simulator='build/default/./SpeciationCounter'*)
Set the location of the coalescence and speciation executables. Make sure that both programs have been compiled for the operating system running the simulations.

Parameters

- **coalescence_simulator** (*str*) – the path to the Coal_v1 executable
- **speciation_simulator** (*str*) – the path to the Speciation_Counter executable

Returns

class Map (*is_sample=None*)
A class for the map object, containing the file name and the variables associated with this map object.

calculate_offset (*file_offset*)
Calculates the offset of the map object from the supplied file_offset.

Parameters **file_offset** (*str/Map*) – the path to the file to calculate the offset. Can also be a Map object with the filename contained.

Returns the offset x and y (at the resolution of the file_home) in integers

calculate_scale (*file_scaled*)
Calculates the scale of map object from the supplied file_scaled.

Parameters **file_scaled** (*str/Map*) – the path to the file to calculate the scale.

Returns the scale (of the x dimension)

check_map ()
Checks that the dimensions for the map have been set and that the map file exists

get_dimensions ()
Calls read_dimensions() if dimensions have not been read, or reads stored information. :return: a list containing [0] x, [1] y, [2] upper left x, [3] upper left y, [4] x resolution, [5] y resolution .. note:: the returned list will contain the x and y offset values instead of the ulx and uly values if the

dimensions have already been set (i.e. self.x_size != 0 and self.y_size != 0)

get_x_y ()
Simply returns the x and y dimension of the file.

Parameters **file** (*str*) – the file to examine

Returns the x and y dimensions

read_dimensions ()
Return a list containing the geospatial coordinate system for the file.

Returns a list containing [0] x, [1] y, [2] upper left x, [3] upper left y, [4] x resolution, [5] y resolution

set_dimensions (*file_name=None*, *x_size=None*, *y_size=None*, *x_offset=None*, *y_offset=None*)
Sets the dimensions and file for the Map object

Parameters

- **file_name** (*str*) – the location of the map object (a csv or tif file). If None, required that file_name is already provided.
- **x_size** (*int*) – the x dimension

- **y_size** (*int*) – the y dimension
- **x_offset** (*int*) – the x offset from the north-west corner
- **y_offset** (*int*) – the y offset from the north-west corner

Returns

set_sample(*is_sample*)
Set the is_sample attribute to true if this is a sample mask rather than an offset map

Parameters **is_sample** (*bool*) – indicates this is a sample mask rather than offset map

setup file

Sets up the NECSim executables. It configures the install by detecting system components and compiles the c++ files, if possible. Command line flags can be provided (see *Compilation Options* for more information) to modify the install.

autoconf()
Runs the *autoconf* bash function (assuming that autoconf is available) to create the *configure* executable.

clean()
Runs make clean in the NECSim directory to wipe any previous potential compile attempts.

configure(*opts=None*)
Runs ./configure –opts with the supplied options. This should create the makefile for compilation, otherwise a RuntimeError will be thrown.

Parameters **opts** – a list of options to pass to the ./configure call

do_compile()
Compiles the c++ NECSim program by running make. This changes the working directory to wherever the module has been installed for the subprocess call.

main(*argv=None*)
Runs the install for compile options provided via the command line, or with default options if no options exist. Running with `-help` or *-h* will display the compilation configurations called from `./configure`.

move_executable(*directory='build/default/'*)
Moves the executable to the specified directory from within NECSim. Will always look for NECSim relative to the setup.py (so can be called from another module location). This allows for multiple compilation options with different versions stored in different folders.

Throws an IOError when the NECSim executable is not found.

Parameters **directory** – the directory to move the NECSim excecutable to.

set_wd()
There should be two calls to this function.

•The first sets the working directory to the location of setup.py

•The second moves the working directory back to its original location.

The second call should be before the move_executable() call so that the new executables are moved to the path supplied relative to the parent file. Otherwise executables will be moved relative to setup.py

hpc_setup file

Sets up the code for the hpc and moves the executable to the correct directory.

build_hpc()
Compiles NECSim with the `--with-hpc`, `--with-verbose` and `--with-fat_tail_dispersal` flags, and moves the executable to ../../Code/ relative to the file location. :return:

6.1.2 C++ Documentation (NECSim)

A Neutral Ecology Coalescence Simulator.

Version: 3.6

Introduction

NECSim

Provided here is a set of tools for running and analysing backwards-time (coalescent) neutral models in ecology. NECSim is a generic spatial coalescence simulator for neutral systems. It applies the model to maps for the supplied parameters and outputs information for each individual to a SQL database.

SpeciationCounter is a program for applying varying speciation rates to outputs of NECSim for analysis after simulations are complete. This enables the main simulation to be run with the *minimum* speciation rate required and afterwards analysis can be completed using different speciation rates. The same functionality is also provided within NECSim for application of speciation rates immediately after simulations are complete.

A Note on the Neutral Theory of Ecology

Neutral theory in ecology refers to the idea that individuals can be modelled as ecologically identical entities, undergoing dispersal, drift and speciation without niche effects or other competitive elements.

Whilst obviously not realistic, the patterns produced by such models can often give a suprisingly accurate protrayal of real-world systems. For more information on the topic, please see *[Hubbell2001]*.

Instructions

Compiling the program

For compilation, there are several provided options:

- Compilation can be handled within PyCoalescence by running `python setup.py`. **This is the recommended option.**
- Alternatively, compilation can be completed with additional options using make. The steps are outlined below
  - You might need to first run `autoconf` from within the NECSim directory to generate the configure executable.
  - Run `./configure` (located within the NECSim directory). Provide additional compilation flags if necessary (detailed below).
  - Run `make all` to create the executable.
  - [Optional] Move the executable (called NECSim) to the *build/Default* directory in **PyCoalescence**.
- If you require compilation outside of the PyCoalescence module, make use of the the file **Makefile** located in **Makefiles/SimpleCompile**. This can be modified and run using `make` to generate the executable.

Compilation on High-performance clusters will likely require an icc compiler and custom linking to the required libraries.

See the Requirements section for a full list of the necessary prerequisites.

Requirements

- The SQLite library available here.
- The Boost library available here.
- C++ compiler (such as GNU g++) with C++11 support.

- Access to the relevant folders for Default simulations (see FAQS).

Recommended, but not essential:

- gdal library available here: provides reading of tif files.
- **The fast-cpp-csv-parser by Ben Strasser, available** here: provides much faster csv read and write capabilities.

Compiler Options

Recognised compiler options include:

Option	Description
–with-debug	Adds additional debugging information
–with-restrict_self	Restricts self-selection during dispersal. Individuals can therefore not become their own parent
–with-verbose	Prints information to the terminal. Otherwise, prints to a log file in *Logs/*
–with-normal_dispersal	Choose a normal distribution for dispersal distances
–with-fat_tail_dispersal	Choose a fat-tailed kernel for dispersal distances
–with-infinite_landscape	Map boundaries are ignored and individuals can disperse to anywhere.
–with-gdal=DIR	Define a gdal library at DIR
–with-hpc	Compile ready for HPC, using intel's icpc compilation and a variety of optimisation flags.
–with-boost=DIR	Define a boost library at DIR

Additional c++ compilation flags can be specified by `CPPFLAGS=opts` for additional library paths or compilation options as required.

Note that gdal and fast-cpp-csv-parser availability will be automatically detected and included in the compilation if possible.

Running simulations

As of version 3.1 and above, the routine relies on supplying command line arguments (see below) for all the major simulation variables. Alternatively, supplying a config .txt file and using the command line arguments `./NECSim -c /path/to/config.txt` can be used for parsing command line arguments from the text file.

Command Line Arguments

The following command line arguments are required. This list can be accessed by running "`./NECSim -h`" or `./NECSim -help`

As of version 3.6 and above, the command line options to be specified are:

1. the seed for the simulation.
2. the simulation task (for file reference).
3. the map config file.
4. the output directory.
5. the minimum speciation rate.

6. the dispersal z_fat value.
7. the dispersal L value.
8. the deme size.
9. the deme sample size.
10. the maximum simulation time (in seconds).
11. the lambda value for moving through non-habitat.
12. the temporal sampling file containing generation values for sampling points in time (null for only sampling the present)
13. the minimum number of species known to exist. (Currently has no effect).
14. (and onwards) speciation rates to apply after simulation.

In this format, the map config file and temporal sampling file are as described in *Config Files*.

Alternatively, by specifying the -f flag, (full mode) as the first argument, the program can read in pre-3.6 command line arguments, which are as followed.

1. the task_iter used for setting the seed.
2. the sample grid x dimension
3. the sample grid y dimension
4. the fine map file relative path.
5. the fine map x dimension
6. the fine map y dimension
7. the fine map x offset
8. the fine map y offset
9. the coarse map file relative path.
10. the coarse map x dimension
11. the coarse map y dimension
12. the coarse map x offset
13. the coarse map y offset
14. the scale of the coarse map compared to the fine (10 means resolution of coarse map = 10 x resolution of fine map)
15. the output directory
16. the speciation rate.
17. the dispersal distance (zfat).
18. the deme size
19. the deme sample size (as a proportion of deme size)
20. the time to run the simulation (in seconds).
21. lambda - the relative cost of moving through non-forest
22. the_task - for referencing the specific task later on.
23. the minimum number of species the system is known to contain.

24. the pristine fine map file to use
25. the pristine coarse map file to use
26. the rate of forest change from pristine
27. the time (in generations) since the pristine forest was seen.
28. the dispersal L value (the width of the kernel.
29. the sample mask, with binary 1:0 values for areas that we want to sample from. If this is not provided then this will default to mapping the whole area.
30. the link to the file containing every generation that the list should be expanded. This should be in the format of a list.
31. (and onwards) - speciation rates to apply after the simulation is complete.

Warning: This method of running simulations is provided for legacy purposes only, and is no longer recommended. For increase functionality, use the condensed command-line format, or use switch to using config files.

Config Files

There are three separate config files which are used when setting up simulations.

- ***main simulation config file*** Contains the main simulation parameters, including dispersal parameters, speciation rates, sampling information and file referencing information. It also includes the paths to the other config files, which must be specified if the main simulation config is used.
- ***map config file*** Contains the map parameters, including paths to the relevant map files, map dimensions, offsets and scaling. This option cannot be null (map dimensions at least must be specified).
- ***time config file*** Contains the temporal sampling points, in generations. If this is 'null', sampling will automatically occur only at the present (generation time=0)

When running the simulation using config files, the path to the *main simulation config file* should be specified, e.g `./NECSim -c /path/to/main/config.txt`.

Main Config File

The configuration containing the majority of the simulation set up, outside of map dimensions. An example file is shown below. This file can be automatically generated by *`create_config()`* in PyCoalescence. An example of this configuration is given below:

```
[main]
job_num = 6
job_type = 6
map_config = output/mapconf.txt
output_directory = output
min_spec_rate = 0.5
zfat = 4
lval = 4
deme = 1
sample_size = 0.1
max_time = 1
lambda = 1
time_config = output/tempconf.txt
```

```
min_species = 1

[spec_rates]
spec_rate1  = 0.6
spec_rate2  = 0.8
```

Map Config File

The map config contains the information for setting up all maps required by the simulation. This involves maps at all times and at all scales. An example is given below.

```
[sample_grid]
path = null
x = 13
y = 13
mask = null

[fine_map]
path = sample/SA_sample_fine.tif
x = 13
y = 13
x_off = 0
y_off = 0

[coarse_map]
path = sample/SA_sample_coarse.tif
x = 35
y = 41
x_off = 11
y_off = 14
scale = 1.0

[pristine_fine0]
path = sample/SA_sample_fine_pristine1.tif
number = 0
time = 1
rate = 0.5

[pristine_coarse0]
path = sample/SA_sample_coarse_pristine1.tif
number = 0
time = 1
rate = 0.5

[pristine_fine1]
path = sample/SA_sample_fine_pristine2.tif
number = 1
time = 4
rate = 0.7

[pristine_coarse1]
path = sample/SA_sample_coarse_pristine2.tif
number = 1
time = 4
rate = 0.7
```

Note: The rates and times between the pairs of pristine fine maps and pristine coarse maps must match up. Without matching values here, there could be undetermined errors, or coarse map values being ignored.

Note: Pristine maps assume the same dimensions as their respective present-day equivalents.

Time Config File

The temporal sampling config file (referred to as "time config file") specifies times, in generations, when a full sample according to the sample map should be taken again. An example of this file is given below.

```
[main]
time0 = 0.0
time1 = 1.0
```

Note: For each speciation rate, all biodiversity measures (such as species' abundances and species' richness) will be calculated for each time supplied separately.

Default parameters

To run the program with the default parameters for testing purposes, run with the command line arguments -d or -dl (for the larger default run). Note that this will require access to the following folders relative to the path of the program for storing the outputs to the default runs:

./Default

./Default/SQL_data/

Outputs

Upon successful completion of a simulation, the two files are created.

- A csv file is created called *Data_{the_task}_{the_seed}.csv* where the_seed and the_task are the values provided in simulation set-up. This contains basic simulation information for quick reference.
- an SQLite database file in the output directory in a folder called *SQL_data*. This database contains all important simulation data over several tables, which can be accessed using a program like DB Browser for SQLite or Microsoft Access. Alternatively, most programming languages have an SQLite interface (RSQlite, python sqlite3)

The tables in the SQLite database are - SIMULATION_PARAMETERS

contains the parameters the simulation was performed with for referencing later.

- SPECIES_LIST

  contains the locations of every coalescence event. This is used by SpeciationCounter to reconstruct the coalescence tree for application of speciation rates after simulations are complete.

- SPECIES_ABUNDANCES

contains the species abundance distributions for each speciation rate and time point that has been specified.

- SPECIES_LOCATIONS [optional]

  contains the x, y coordinates of every individual at each time point and for every specified speciation rate, along with species ID numbers.

- FRAGMENT_ABUNDANCES [optional]

  contains the species abundance distributions for each habitat fragment, either specified by the fragment csv file, or detected from squares across the map.

Additional information can be found in *SpeciationCounter* regarding the optional database tables.

SpeciationCounter

SpeciationCounter provides a method for applying additional speciation rates to outputs from NECSim, without having to re-run the entire simulation. SpeciationCounter works by reconstructing the coalescence tree, checking at each point if an additional speciation rate has occured. As such, SpeciationCounter can only apply speciation rates higher than the initial speciation rate the program was run with.

Applying Speciation Rates

Run `./SpeciationCounter` with the following command-line options:

1. path to the SQLite database file (this is the output of a NECSim simulation).
2. T/F for recording spatial data. If true, the SPECIES_LOCATIONS table will be created (see *Outputs*.)
3. a sample mask to use for the data. Species' identities for the individuals will only be calculated from locations specified by a 1 in the sample mask (0 otherwise). Use "null" to record all locations.
4. a *time config file* specifying temporal sampling locations.
5. T/F for calculating fragment species abundances individually. If true, the FRAGMENT_ABUNDANCES table will be created containing the species abundances for each fragment calculated as a square of continuous habitat. Alternatively, can specify a csv file that contains the fragment information in the following format. All x, y coordinates are given on the sample grid size specified at simulation run-time.

```
fragment_name1, x_min, y_min, x_max, y_max, number_of_individuals
fragment_name2, x_min, y_min, x_max, y_max, number_of_individuals
...
```

6. A speciation rate to apply. Can list multiple speciation rates by supplying arguments 7 onwards.

Debugging

Most errors will return an error code in the form "ERROR_NAME_XXX: Description" a list of which can be found in ERROR_REF.txt.

Brief Class Descriptions

A brief description of the important classes is given below. Some classes also contain customised exceptions for better tracing of error handling.

- The *Tree* class
  - The most important class!
  - Contains the main setup, run and data output routines.
  - *setup()* imports the data files from csv (if necessary) and creates the in-memory objects for the storing of the coalescence tree and the spatial grid of active lineages. Setup time mostly depends on the size of the csv file being imported.
  - Run continually loops over sucessive coalesence, move or speciation events until all individuals have speciated or coalesced. This is where the majority of the simulation time will be, and is mostly dependent on the number of individuals, speciation rate and size of the spatial grid.
  - At the end of the simulation, the sqlCreate() routine will generate the in-memory SQLite database for storing the coalescent tree. It can run multiple times if multiple speciation rates are required. *outputData()* will then be called to create a small csv file containing important information, and output the SQLite database to file if required.
- The *Treenode* class
  - Contains a single record of a node on the phylogenetic tree, to be used in reassembling the tree structure at the end of the simulation.
- The *Datapoint* class
  - Contains a single record of the location of a lineage.
- The *NRrand* class
  - Contains the random number generator, as written by James Rosindell (j.rosindell@imperial.ac.uk).
- The *Map* class
  - Contains the routines for importing and calling values from the map objects.
  - The *getVal()* and *runDispersal()* functions can be modified to produce altered dispersal behaviour, or alterations to the structure of the *Row*
- The *Matrix* and *Row* classes
  - Based on code written by James Rosindell (j.rosindell@imperial.ac.uk).
  - Handles indexing of the 2D object plus importing values from a csv file.
- The *SpeciesList* class
  - Contains the list of individuals, for application in a matrix, to essentially create a 3D array.
  - Handles the positioning of individuals in space within a grid cell.
- The *ConfigOption* class
  - Contains basic functions for importing command line arguments from a config file, providing an alternative way of setting up simulations.
- The *Treelist* class
  - Provides the routines for applying different speciation rates to a phylogenetic tree, to be used either immediately after simulation within NECSim, or at a later time using *SpeciationCounter*

Known Bugs

- Simulation pause and resume functions do not work properly at this time (should be fixed in a later update).

- Simulations run until completion, rather than aiming for a desired number of species. This is an intentional change. Functions related to this functionality remain but are deprecated.
- In SpeciationCounter, only continuous rectangular fragments are properly calculated. Other shapes must be calculated by post-processing.
- In SpeciationCounter, 3 fragments instead of 2 will be calculated for certain adjacent rectangular patches.

FAQS (WIP)

- **Why doesn't the default simulation output anything?**
  - Check that the program has access to the folders relative to the program at *Default/*
- **Why can't I compile the program?**
  - This could be due to a number of reasons, most likely that you haven't compiled with access to the lsqlite3 or boost packages. Installation and compilation differs across different systems; for most UNIX systems, compiling with the linker arguments -lsqlite3 -lboost_filesystem and -lboost_system will solve problems with the compiler not finding the sqlite or boost header file.
  - Another option could be the potential lack of access to the fast-cpp-csv-parser by Ben Strasser, available here. If use_csv has been defined at the head of the file, try without use_csv or download the csv parser and locate the folder within your working directory at compilation.
- **Every time the program runs I get error code XXX.**
  - Check the ERROR_REF.txt file for descriptions of the files. Try compiling with the *DEBUG* precursor to gain more information on the problem. It is most likely a problem with the set up of the map data (error checking is not yet properly implemented here).

Contacts

Author: **Samuel Thompson**

Contact: samuelthompson14@imperial.ac.uk - thompsonsed@gmail.com

Institution: Imperial College London and National University of Singapore

Based heavily on code by **James Rosindell**

Contact: j.rosindell@imperial.ac.uk

Institution: Imperial College London

Licence

Class Hierarchy

- *Struct Config_Exception*
- *Struct Fatal_Exception*
- *Struct Fragment*

- *Struct Main_Exception*
- *Struct Map_Exception*
- *Struct Map_Fatal_Exception*
- *Struct Mapvars*
- *Struct SectionOption*
- *Struct SpeciesException*
- *Class ConfigOption*
- *Class Datamask*
- *Class Datapoint*
- *Class Map*
- *Class Matrix*
- *Class NRrand*
- *Class Row*
- *Class Samplematrix*
- *Class simParameters*
- *Class SpeciesList*
- *Class Tree*
- *Class Treelist*
- *Class Treenode*

File Hierarchy

- *File Config.cpp*
- *File Config.h*
- *File CustomExceptions.h*
- *File Datapoint.h*
- *File Fattaildeviate.h*
- *File main.cpp*
- *File Map.cpp*
- *File Map.h*
- *File Matrix.h*
- *File Setup.cpp*
- *File Setup.h*
- *File SpeciationCounter.cpp*
- *File SpeciesList.h*
- *File Tree.h*
- *File Treelist.cpp*

- *File Treelist.h*
- *File Treenode.h*

Full API

Namespaces

Namespace std

Classes and Structs

Struct Config_Exception

- Defined in *File CustomExceptions.h*

struct `Config_Exception`
A structure for all exceptions thrown within config processes.

Inherits from runtime_error

Public Functions

`Config_Exception`()

`Config_Exception`(string *msg*)

Struct Fatal_Exception

- Defined in *File CustomExceptions.h*

struct `Fatal_Exception`
This is called any time a fatal exception is called and the program is unwound and ended.

Inherits from runtime_error

Public Functions

`Fatal_Exception`()

`Fatal_Exception`(string *msg*)

Struct Fragment

- Defined in *File Treelist.h*

struct `Fragment`
Contains the information needed for defining a fragment. Fragments can be detected from the *Samplematrix* object (which only detects rectangular fragments), or (preferably) is read from an input file. Currently all fragments must be rectangular, although they can be larger than the intended shape if necesssary.

Public Members

string **name**

unsigned long **x_east**

unsigned long **x_west**

unsigned long **y_north**

unsigned long **y_south**

unsigned long **num**

double **area**

Struct Main_Exception

- Defined in *File CustomExceptions.h*

struct Main_Exception

These are used for non-fatal exception thrown from within the main simulation where no-more specific location information is possible.

Inherits from runtime_error

Public Functions

Main_Exception()

Main_Exception(string *msg*)

Struct Map_Exception

- Defined in *File CustomExceptions.h*

struct Map_Exception

The non-fatal exception thrown when a problem is encountered in any *Map* object processes.

Inherits from runtime_error

Public Functions

Map_Exception()

Map_Exception(string *msg*)

Struct Map_Fatal_Exception

- Defined in *File CustomExceptions.h*

struct Map_Fatal_Exception

The fatal exception thrown when a problem is encountered in any *Map* object processes.

Inherits from runtime_error

Public Functions

Map_Fatal_Exception()

Map_Fatal_Exception(string *msg*)

Struct Mapvars

- Defined in *File Map.h*

struct Mapvars

Stores and imports the variables required by the *Map* object. Used to setting the *Map* variables in a more elegant way.

Public Functions

void **import**(**const** vector<string> &*comargs*, bool *fullmode*)

Imports the *Map* variables from a vector of command line arguments.

Parameters

- comargs:
- fullmode: whether the full import mode has been specified.

void **setPristine**(unsigned int *n*)

Alters the pristine parameters to the configuration matching the input number. If no configuration option exists for this number, bPristine will be set to true.

Parameters

- n:

void **printVars**()

Prints selected important variables to the terminal.

Public Members

string **finemapfile**

string **coarsemapfile**

string **outdirectory**

string **pristinefinemapfile**

string **pristinecoarsemapfile**

string **samplemaskfile**

long **the_task**

unsigned long **vargridxsize**

unsigned long **vargridysize**

unsigned long **varfinemapxsize**

unsigned long **varfinemapysize**

unsigned long **varfinemapxoffset**

unsigned long **varfinemapyoffset**

unsigned long **varcoarsemapxsize**

unsigned long **varcoarsemapysize**

unsigned long **varcoarsemapxoffset**

unsigned long **varcoarsemapyoffset**

unsigned long **varcoarsemapscale**

unsigned long **desired_specnum**

double **lambda**

long **deme**

double **deme_sample**

long double **spec**

double **zfat**

double **maxtime**

double **dPristine**

double **dForestTransform**

double **dLvalue**

string **autocorrel_file**

ConfigOption **configs**

bool **bPristine**

Friends

ostream &**operator<<** (ostream &*os*, **const** *Mapvars* &*m*)
Overloading the << operator for outputting to the output stream.

Return os the output stream.

Parameters

- `os`: the output stream.
- `m`: the *Mapvars* object.

istream &**operator>>** (istream &*is*, *Mapvars* &*m*)
Overloading the >> operator for inputting from an input stream.

Return is the input stream

Parameters

- `is`: the input stream

- m: the mapvars object

Struct SectionOption

- Defined in *File Config.h*

struct SectionOption
A simple container for importing options from a config file.

Public Functions

string **getOption**(string *refval*)
Returns the value for the provided reference from within the key.

Return the requested value as a string. Returns string "null" if no reference is found.

Parameters

- refval: the reference to obtain the value of

Public Members

string **section**

vector<string> **val**

vector<string> **refs**

Friends

ostream &**operator<<**(ostream &*os*, **const** *SectionOption* &*k*)
Overloading the << operator for outputting to the output stream.

Return os the output stream.

Parameters

- os: the output stream.
- k: the KeyOption object.

istream &**operator>>**(istream &*is*, *SectionOption* &*k*)
Overloading the >> operator for inputting from an input stream.

Return is the input stream

Parameters

- is: the input stream
- k: the KeyOption object

Struct SpeciesException

- Defined in *File CustomExceptions.h*

struct `SpeciesException`
An exception thrown whenever a non-fatal Species exception is thrown.

Inherits from runtime_error

Public Functions

`SpeciesException`()
Throws a runtime_error with a custom message indicating source.

`SpeciesException`(string *msg*)
Overloaded runtime_error call which provides error message parsing.

Parameters

- `msg`: the message to be passed to the runtime_error

Class ConfigOption

- Defined in *File Config.h*

class `ConfigOption`
Config option class to store and import options from a file.

Public Functions

`ConfigOption`()
default construtor for *ConfigOption*

string **`getFile`**()
Standard getter for the file name.

Return string the file name.

bool **`getMain`**()
Gets if the config is a main command-line import.

Return bool bMain

bool **`getFull`**()
Gets if the config requires each KeyOption structure to be returned after each read.

Return bool bFullParse

void **`setConfig`**(string *file*, bool *main*, bool *full_parse* = false)
Sets the config file the specified string. A boolean is also provided, set equal to true if this is the main command line import. This causes the deletion of the first few command line options after import.

Parameters

- `file`: the target config file (in .txt format).
- `main`: boolean of if this is the main command line import.
- `full_parse`: sets bFullParse to provided value

void **parseConfig**()
Reads a config file of a specific configuration. Each line must be a set of values related to the object determined by ref='object'. Each key must have the form key=value, which will be read as a string into a KeyOption structure.

void **testPrint**()

vector<*SectionOption*> **getSectionOptions**()
Returns the vector of key options imported from the file.

Return vector of key options

SectionOption **operator[]**(int *index*)

unsigned int **getSectionOptionsSize**()
Gets the size of the key options vector.

Return the size of the configuration vector.

vector<string> **getSections**()
Gets the sections contained in the SectionOptions object.

Return A vector of the section names.

bool **hasSection**(string *sec*)
Checks whether the config option has the specified section.

Return true if the section has been found

Parameters

- `sec`: the section name to check for

vector<string> **getSectionValues**(string *sec*)
Gets all values within a section.

Throws a *Config_Exception* if the section is not found.

Return a vector of the section's values.

Parameters

- `sec`: the section to find values for

string **getSectionOptions**(string *section*, string *ref*)
Returns a specific value for a particular key options and reference.

Return the string at the correct place in KeyOptions.val

Parameters

- `key`: the key to match (this corresponds to the line the value comes from
- `ref`: the reference to match

int **importConfig**(vector<string> &*comargs*)
Imports the parameters from the config file and returns an integer of the number of arguments.

Return a count of the number of arguments (should also be the size of comargs).

Parameters

- `comargs`: a vector of command line arguments to import to from file.

Friends

ostream &**operator<<**(ostream &*os*, **const** *ConfigOption* &*c*)
Overloading the << operator for outputting to the output stream.

Return os the output stream.

Parameters

- `os`: the output stream.
- `c`: the *ConfigOption* object.

istream &**operator>>**(istream &*is*, *ConfigOption* &*c*)
Overloading the >> operator for inputting from an input stream. Note that the config file must still exist for re-inport and parsing.

Return is the input stream

Parameters

- `is`: the input stream
- `c`: the *ConfigOption* object

Class Datamask

- Defined in *File Map.h*

class Datamask
Contains the *Datamask* object, a *Matrix* of booleans describing the spatial sampling pattern.

Public Functions

Datamask()
The *Datamask* constructor.

void **importDatamask**(*Mapvars* &*mapvarin*)
Imports the *Datamask* from the supplied csv file. Requires the mapvar object for command line parsing.

Parameters

- `mapvarin`: the *Mapvars* object to parse.

bool **getVal**(long *x*, long *y*)
Calculates the matrix value at the provided x, y location. If everywhere is sampled, simply returns true, as no sample_mask will be stored in memory. This is to save RAM where possible.

Return the sample_mask value at x,y (or true if the file was "null").

Parameters

- `x`: the x position on the grid.
- `y`: the y position on the grid.

Public Members

Matrix<bool> **sample_mask**
A binary grid telling whether or not the cell should be sampled.

Class Datapoint

- Defined in *File Datapoint.h*

class Datapoint
A data object used in coalescence simulations for calculating the output. Data from this object is outputted to an SQLite database after simulations are complete.

Author Samuel Thompson

Date 30/08/2016

Public Functions

Datapoint()
Standard constructor.

~Datapoint()
Standard destructor.

void **setup**(unsigned long *x*, unsigned long *y*, long *xwrapin*, long *ywrapin*, unsigned long *matrix_position*, unsigned short *listposin*, double *dMinmaxin*)
Setup of lineage data with any information that's wanted. Note that nwrap is set to 0 in this routine.

Parameters

- `x`: the x position on the grid
- `y`: the y position on the grid
- `xwrapin`: the number of wraps of the location on the grid in the x direction.
- `ywrapin`: the number of wraps of the location on the grid in the y direction.
- `matrix_position`: the position in the *Treenode* reference object.
- `listposin`: the position within the *SpeciesList* object at the relevant x,y position.
- `dMinmaxin`: the input maximum minimum speciation rate required for speciation to have occured on this branch.

void **setup**(*Datapoint datin*)
Copy constructor from another *Datapoint* object.

Parameters

- `datin`: a *Datapoint* object to copy the data from.

void **setMpos**(unsigned long `z`)
Sets the mpos (the position within the *Row* of *Treenode* objects.).

Parameters

- `z`: the desired mpos.

void **setNext**(unsigned long *x*)
Set the next link in the linked list.

Parameters

- `x`: the next *Datapoint* object.

void **setListpos**(unsigned short *l*)
Sets the list position within the *SpeciesList* object.

Parameters

- `l`: the input list position.

void **setNwrap**(unsigned short *n*)
Sets the number of wraps from the first *SpeciesList* wrapped lineage. If and only if this is 0, the lineage is within the main grid (i.e xwrap and ywrap should be 0).

Parameters

- `n`: the desired nwrap.

void **setMinmax**(double *d*)
Sets the minmax variable. This is the minimum maximum speciation rate required for speciation to have occured on this branch.

Parameters

- `d`: the minmax to set.

void **subtractListpos**()
Decreases the list position by one.

unsigned long **getXpos**()
Get the x position.

Return the xpos.

unsigned long **getYpos**()
Get the y position.

Return the ypos.

long **getXwrap**()
Get the x wrapping.

Return the xwrap.

long **getYwrap**()
Get the y wrapping.

Return the ywrap.

unsigned long **getMpos**()
Get the matrix position variable.

Return the mpos.

unsigned long **getNext**()

unsigned short **getListpos**()
Gets the list position with the *SpeciesList* object at the relevant x,y position.

Return the listpos.

unsigned short **getNwrap**()
Get the position in the linked list from the *SpeciesList* object. If this is 0, indicates the lineage lies on the original grid, and xwrap and ywrap should be 0.

Return the nwrap.

double **getMinmax**()
Get the maximum minimum speciation rate required for speciation to have occured on this branch.

Return the minmax.

void **decreaseNwrap**()
Decreases the nwrap by 1 (to a minimum of 0).

void **setEndpoint**(long *x*, long *y*, long *xwrapin*, long *ywrapin*)
Sets the position in space.

Parameters

- `x`: the x position.
- `y`: the y position.
- `xwrapin`: the number of wraps in the x direction.
- `ywrapin`: the number of wraps in the y direction.

Friends

ostream &**operator<<**(ostream &*os*, **const** *Datapoint* &*d*)
An operator for piping the variables of the *Datapoint* object to the output stream.

Return returns the output stream at the end.

Parameters

- `os`: the output stream.
- `d`: the *Datapoint* object to output.

istream &**operator>>**(istream &*is*, *Datapoint* &*d*)
An operator for piping the variables in to the *Datapoint* object from the input stream.

Return returns the input stream at the end.

Parameters

- `is`: the input stream
- `d`: the *Datapoint* object to input to.

Class Map

- Defined in *File Map.h*

class Map
Contains all maps and provides the functions for accessing a grid cell in the correct temporal and spacial location. The function *runDispersal()* also provides the move routine, provided two alternative methods for moving individuals.

Public Functions

Map()
The default constructor.

void **setDims**(*Mapvars mapvarsin*)
Sets the dimensions of the grid, the area where the species are initially sampled from. This function must be run before any of the calc map functions to allow for the correct deme allocation.

Parameters

- `mapvarsin`: the *Mapvars* object containing the map variables to import

bool **checkMapExists**()

void **calcFineMap**()
Imports the fine map object from file and calculates the correct values at each point. Without a map to input, the fine map will simply be a matrix of 1s.

void **calcPristineFineMap**()
Imports the pristine fine map object from file and calculates the correct values at each point. Without a map to input, the pristine fine map will simply be a matrix of 1s. This has the potential to be changed easily in future versions.

void **calcCoarseMap**()
Imports the coarse map object from file and calculates the correct values at each point. Without a map to input, the coarse map will simply be a matrix of 1s. This has the potential to be changed easily in future versions.

void **calcPristineCoarseMap**()
Imports the pristine coarse map object from file and calculates the correct values at each point. Without a map to input, the pristine coarse map will simply be a matrix of 1s. This has the potential to be changed easily in future versions.

void **setTimeVars**(double *dPristinein*, double *dForestTransformin*)
Sets the time variables.

Parameters

- `dPristinein`: the time (in generations) since a pristine habitat state was achieved.
- `dForestTransformin`: the rate of transform of the forest up until the pristine time. A value of 0.2 would mean 20% of the change occurs linearlly up until the pristine time and the remaining 80% occurs in a jump to the pristine state.

void **calcOffset**()
Calculates the offset and extremeties of the fine map.

Note that setting lambda to a value other than 1 can massively increase simulation time.

void **validateMaps**()
Checks that the map file sizes are correct and that each value on the fragmented maps is less than the pristine maps. This should be disabled in simulations where habitat sizes are expected to shrink as well as grow.

void **updateMap**(double *generation*)
Updates the maps to the newer map.

bool **isPristine**()
Gets the pristine boolean.

Return the pristine map state.

void **setPristine**(**const** bool &*bPristinein*)
Sets the pristine state of the system.

Parameters

- `bPristinein`: the pristine state.

unsigned short **getVal**(**const** double &*x*, **const** double &*y*, **const** long &*xwrap*, **const** long &*ywrap*, **const** double &*dCurrentGen*)
Gets the value at a particular coordinate from the correct map. Takes in to account temporal and spatial referencing. This version assumes an infinite landscape.

Return the value on the correct map at the correct space.

Parameters

- `x`: the x position on the grid.
- `y`: the y position on the grid.

- `xwrap`: the number of wraps in the x dimension..
- `ywrap`: the number of wraps in the y dimension..
- `dCurrentGen`: the current generation time.

unsigned long **`getInitialCount`**(double *dSample*, *Datamask* &*samplemask*)
Counts the number of spaces available in the initial species space. Requires the samplemask to check the sampling area.

Return the total number of individuals predicted to initially exist on the map.

Parameters

- `dSample`: the sample proportion (from 0 to 1).
- `samplemask`: the *Datamask* object to sample from.

bool **`checkMap`**(**const** double &*x*, **const** double &*y*, **const** long &*xwrap*, **const** long &*ywrap*, **const** double *generation*)
Checks whether the point is forest or non-forest.

Return a boolean of whether the map is forest or non-forest.

Parameters

- `x`: the x position on the grid.
- `y`: the y position on the grid.
- `xwrap`: the number of wraps in the x dimension.
- `ywrap`: the number of wraps in the y dimension.
- `generation`: the current generation time.

bool **`checkFine`**(**const** double &*x*, **const** double &*y*, **const** long &*xwrap*, **const** long &*ywrap*)
Checks whether the point comes from the fine grid.

Return a boolean of whether the location is on the fine map.

Parameters

- `x`: the x position.
- `y`: the y position.
- `xwrap`: the number of wraps in the x dimension.
- `ywrap`: the number of wraps in the y dimension.

void **`convertCoordinates`**(double &*x*, double &*y*, long &*xwrap*, long &*ywrap*)
Converts the coordinates to within the original grid, altering the xwrap and ywrap consequently.

Parameters

- `x`: the x position.
- `y`: the y position.
- `xwrap`: the number of wraps in the x dimension.
- `ywrap`: the number of wraps in the y dimension.

void **runDispersal**(**const** double &*dist*, **const** double &*angle*, long &*startx*, long &*starty*, long &*startxwrap*, long &*startywrap*, bool &*disp_comp*, **const** double &*generation*)

The function that actually performs the dispersal. It is included here for easier programming and efficiency as the function doesn't need to perform all the checks until the edge of the fine grid.

Parameters

- `dist`: the distance travelled (or "distance energy" if lambda is not 1).
- `angle`: the angle of movement.
- `startx`: the start x position.
- `starty`: the start y position.
- `startxwrap`: the start number of wraps in the x dimension.
- `startywrap`: the start number of wraps in the y dimension.
- `disp_comp`: a boolean of whether the dispersal was complete or not. This value is returned true if dispersal is to habitat, false otherwise.
- `generation`: the time in generations since the start of the simulation.

void **printVars**()

Prints some selected *Map* variables to the terminal.

void **clearMap**()

Wipes the map of all variables. Only really useful for testing purposes.

Friends

ostream &**operator<<**(ostream &*os*, **const** *Map* &*r*)

Operator for outputting the *Map* object variables to an output stream. This is used for storing the *Map* object to file.

Return the output stream.

Parameters

- `os`: the output stream.
- `r`: the *Map* object to output.

istream &**operator>>**(istream &*is*, *Map* &*r*)

Operator for inputting the *Map* object variables from an input stream. This is used for reading the *Map* object from file.

Return the input stream.

Parameters

- `is`: the input stream.
- `r`: the *Map* object to input to.

Class Matrix

- Defined in *File Matrix.h*

template <class *T*>
class **Matrix**

A class containing the *Matrix* object, set up as an array of *Row* objects. Includes basic operations, as well as the *importCsv()* function for more advanced reading from file.

Public Functions

Matrix(int *rows* = 0, int *cols* = 0)

The standard constructor.

Parameters

- `rows`: optionally provide the number of rows.
- `cols`: optionally provide the number of columns.

Matrix(**const** *Matrix* &*m*)

The copy constructor.

Parameters

- `m`: a *Matrix* object to copy from.

~Matrix()

The descructor.

void **SetSize**(int *rows*, int *cols*)

Sets the matrix size. Similar concept to that for Rows.

Parameters

- `rows`: the number of rows.
- `cols`: the number of columns.

int **GetCols**() **const**

Getter for the number of columns.

Return the number of columns.

int **GetRows**() **const**

Getter for the number of rows.

Return the number of rows.

Row<T> &**operator[]**(unsigned long *index*)

Overoads the [] operator for *Matrix*. Allows referencing of a value i,j using *Matrix*[i][j]. Includes error checking for if the indices are out of range of the matrix. Note that this functionality has been altered since the original file generation.

Return the matrix row object.

Parameters

- `index`: the row number to get the value from.

Matrix &**operator=**(**const** *Matrix* &*m*)
Overloading the = operator.

Parameters

- `m`: the matrix to copy from.

const *Matrix* **operator+**(**const** *Matrix* &*m*)
Overloading the + operator.

Return the matrix object which is the sum of the two matrices.

Parameters

- `m`: the matrix to add to this matrix.

const *Matrix* **operator-**(**const** *Matrix* &*m*)
Overloading the - operator.

Return the matrix object which is the subtraction of the two matrices.

Parameters

- `m`: the matrix to subtract from this matrix.

Matrix &**operator+=**(**const** *Matrix* &*m*)
Overloading the += operator so that the new object is written to the current object.

Parameters

- `m`: the *Matrix* object to add to this matrix.

Matrix &**operator-=**(**const** *Matrix* &*m*)
Overloading the -= operator so that the new object is written to the current object.

Parameters

- `m`: the *Matrix* object to subtract from this matrix.

const *Matrix* **operator***(**const** double *s*)
Overloading the * operator for scaling.

Return the scaled matrix.

Parameters

- `s`: the constant to scale the matrix by.

const *Matrix* **operator***(*Matrix* &*m*)
Overloading the * operator for matrix multiplication. Multiplies each value in the matrix with its corresponding value in the other matrix.

Return the product of each ith,jth value of the matrix.

Parameters

- m: the matrix to multiply with

bool **symmetric**()
Checks if the matrix is symmetric or not.

Return true if matrix is symmetric, false otherwise.

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** double &*t*)
Overloaded function for setting the (double) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works.

Parameters

- x: the x index.
- y: the y index.
- value: the value to set
- t: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** char &*t*)
Overloaded function for setting the (char) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works. Note this version of the function has not been properly tested and is included for compilation purposes only.

Parameters

- x: the x index.
- y: the y index.
- value: the value to set
- t: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** int8_t &*t*)
Overloaded function for setting the (int8_t) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works. Note this version of the function has not been properly tested and is included for compilation purposes only.

Parameters

- x: the x index.
- y: the y index.
- value: the value to set
- t: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** uint8_t &*t*)
Overloaded function for setting the (uint8_t) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works. Note this version of the function has not been properly tested and is included for compilation purposes only.

Parameters

- `x`: the x index.
- `y`: the y index.
- `value`: the value to set
- `t`: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** bool &*t*)
Overloaded function for setting the (integer) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works.

Parameters

- `x`: the x index.
- `y`: the y index.
- `value`: the value to set
- `t`: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** unsigned long &*t*)
Overloaded function for setting the (unsigned long integer) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works.

Parameters

- `x`: the x index.
- `y`: the y index.
- `value`: the value to set
- `t`: the template class (passed to the function).

void **setValue**(**const** unsigned long &*x*, **const** unsigned long &*y*, char **value*, **const** long &*t*)
Overloaded function for setting the (long integer) value of a matrix based on the template type of the matrix. This appears like a bit of a goofy workaround and there might be a better way of doing this, but it works.

Parameters

- `x`: the x index.
- `y`: the y index.
- `value`: the value to set
- `t`: the template class (passed to the function).

void **import**(string *filename*)
Imports the matrix from either a csv or tif file. Calls either *importCsv()* or importTif() dependent on the provided file type.

Parameters

- `filename`: the file to import.

void **importCsv**(string *filename*)
Imports the matrix from a tif file using the gdal library functions.

Parameters

- `filename`: the path to the file to import. Imports the matrix from a csv file using the fast-csv-parser method.
- `filename`: the path to the file to import. Imports the matrix from a csv file using the standard, slower method.

Parameters

- `filename`: the path to the file to import.

Protected Attributes

unsigned long **numCols**

unsigned long **numRows**

Row<T> ***matrix**

Friends

ostream &**operator<<**(ostream &*os*, **const** *Matrix* &*m*)
Overloading the << operator for outputting to an output stream. This can be used for writing to console or storing to file.

Return the output stream.

Parameters

- `os`: the output stream.
- `m`: the matrix to output.

istream &**operator>>**(istream &*is*, *Matrix* &*m*)
Overloading the >> operator for inputting from an input stream. This can be used for writing to console or storing to file.

Return the input stream.

Parameters

- `is`: the input stream.
- `m`: the matrix to input to.

Class NRrand

- Defined in *File Fattaildeviate.h*

class NRrand
Contains the functions for random number generation.

Public Functions

NRrand()
Standard constructor.

void **setSeed**(long *seed*)
Sets the seed to the given input. Is only seeded if the seed hasn't already been provided.

Parameters

- `seed`: the input seed.

double **d01**()
The random number generator. Uses Schrage's method and a shuffle table to generate the output.

Return the random number (a double between 0 and 1).

long **i0**(long *max*)
Generates a random number uniformly from 0 to the maximum value provided.

Return an integer of the produced random number.

Parameters

- `max`: the maximum number.

double **norm**()
Generates a normally distributed number Uses the standard normal distribution.

Return the random number from a normal distribution.

void **setDispersalParams**(double *zfatin*, double *dLvaluein*)
Sets the dispersal parameters, avoiding requirement to provide these numbers each function call. This is only relevant for fat-tailed dispersal calls.

Parameters

- `zfatin`: the fatness of the fat-tailed dispersal kernel.
- `dLvaluein`: the width of the fat-tailed dispersal kernel.

double **fattail**(double *z*)
Call from the fat-tailed dispersal kernel with the provided zfat. This is the original version.

Return a random number drawn from the fat-tailed dispersal kernel.

Parameters

- `z`: the desired zfat.

double **fattail**()
Call from fat-tailed dispersal kernel. This function requires *setDispersalParams()* has already been called.

Return a random number drawn from the fat-tailed dispersal kernel.

double **direction**()
An old direction-producing function.

Return the direction.

bool **event**(double *probin*)
For a given event probability, returns the probability that the event has occured.

Return whether or not the event has occured.

Parameters

- `probin`: the event probability.

Friends

ostream &**operator<<**(ostream &*os*, **const** *NRrand* &*r*)
Outputs the *NRrand* object to the output stream. Used for saving the object to file.

Return the output stream.

Parameters

- `os`: the output stream.
- `r`: the *NRrand* object to output.

istream &**operator>>**(istream &*is*, *NRrand* &*r*)
Inputs the *NRrand* object from the input stream. Used for reading the *NRrand* object from a file.

Return the input stream.

Parameters

- `is`: the input stream.
- `r`: the *NRrand* object to input to.

Class Row

- Defined in *File Matrix.h*

template <class *T*>
class Row
Contains a tempate *Row* class and basic operations. Uses an array to store the row.

Public Functions

Row(int *cols* = 0)
Standard constructor.

Parameters

- `cols`: optionally provide the number of rows to initiate with.

~Row()
Standard destructor.

Row(**const** *Row* &*r*)
Copy constructor.

Parameters

- r: the *Row* object to copy from.

void **SetRowSize**(unsigned long *n*)
Setter for the row size.

Parameters

- n: the number of rows to initiate with. *SetRowSize()* deletes any old data, and allocates space for new data, unless we set the number of columns to 0, in which case it merely deletes the data. This lets us use this function for construction, destruction, and dynamic modification in one method.

void **changeSize**(unsigned long *n*)
Changes the size of the array.

Parameters

- n: the new size to change to. Note that no checks are performed that the new row size is larger than the old row size. Thus is this function is used to shrink the row size, a bad_alloc error will likely be thrown.

unsigned long **size**()
Getter for the size of the array.

Return the number of columns.

T &**operator[]**(unsigned long *column*)
Overloading the [] operator to allow for simple referencing.

Return the value in the specified column. Note that different versions deal with values outside of (0,numCols) in different ways.

Note updated to throw an out_of_range exception if the column is out of the row range.

Parameters

- column: the column to get the value from.

Row &**operator=**(**const** *Row* &*r*)
Overloading the = operator to allow for copying data across.

Parameters

- r: the *Row* object to copy data from.

Friends

ostream &**operator<<**(ostream &*os*, **const** *Row* &*r*)
Overloading the << operator for outputting to the output stream.

Return os the output stream.

Parameters

- `os`: the output stream.
- `r`: the *Row* object to output from.

istream &**operator>>**(istream &*is*, *Row* &*r*)
Overloading the << operator for inputting from an input stream.

Return the input stream.

Parameters

- `is`: the input stream.
- `r`: the *Row* object to input to.

Class Samplematrix

- Defined in *File Treelist.h*

class Samplematrix
A child of the *Matrix* class as booleans. Used for determining where to sample species from.

Inherits from *Matrix< bool >*

Public Functions

Samplematrix()
Inherit construction from the *Matrix* class, but also set the booleans.

void **setIsNull**(bool *b*)
Checks whether the object is defined as null or not.

Parameters

- `b`: the boolean to set

bool **getIsNull**()
Getter for the null boolean.

Return the null boolean.

bool **getTestVal**(unsigned int *xval*, unsigned int *yval*)
Returns the value at the x,y position. This is used for testing purposes only.

Return the value at x,y.

Parameters

- `xval`: the x coordinate.
- `yval`: the y coordinate

bool **getVal**(unsigned int *xval*, unsigned int *yval*)
Returns the value at the x,y position. Also checks whether or not the map is set to null, or whether the value comes from within a fragment.

Return the value at x,y.

Parameters

- `xval`: the x coordinate.
- `yval`: the y coordinate

void **setFragment**(*Fragment* *fragment_in*)
Set the fragment for the samplemask to some calculated fragment. This can be set multiple times.

Parameters

- `fragment_in`: the *Fragment* to set the samplemask to.

void **removeFragment**()
Removes the fragment.

Class simParameters

- Defined in *File SpeciationCounter.cpp*

class simParameters
Contains the simulation parameters that are read from the command line.

Public Members

bool **RecordSpatial**

bool **bMultiRun**

bool **RecordFragments**

string **filename**

vector<double> **vSpecRates**

string **samplemask**

string **autocorrel_file**

vector<double> **autocorrel_times**

bool **bAuto**

string **fragment_config_file**

Class SpeciesList

- Defined in *File SpeciesList.h*

class `SpeciesList`

Contains a list of the species that exist at one location. The *Row* object, list, contains the active reference number, for looking up the lineage in a *Row* of *Datapoint* objects. Also contains the functions for correctly generating coalescence probabilities and list management.

Note that the maximum size of the list is constrained by the maximum size of unsigned short. If problems are encountered with this, a fairly simple process of replacing these procedures with unsigned longs is possible but would increase RAM usage.

Public Functions

`SpeciesList`()

Default constructor.

`~SpeciesList`()

Default destructor.

void **`fillList`**()

Fills the list with 0, up to the specified maximum size.

void **`initialise`**(unsigned short *maxsizein*)

Initialises the list to the specified size.

Parameters

- `maxsizein`: the maximum list size.

void **`setMaxsize`**(unsigned short *maxsizein*)

Sets the maxsize without altering the actual size of list.

Parameters

- `maxsizein`: The new maximum size to set.

void **`setSpecies`**(int *index*, unsigned long *new_val*)

Set specific entry to a particular species reference number.

Parameters

- `index`: the location in list of the species.
- `new_val`: the new species reference to set list[index] to.

void **`setNext`**(unsigned long *n*)

Set the next active lineage (for wrapping purposes).

Parameters

- `n`: the lineage to set as the first wrapped lineage.

void **`setNwrap`**(unsigned short *nr*)

Set the number of wrapping lineages.

Parameters

- `nr`: the number of wrapped lineages.

unsigned short **addSpecies**(unsigned long *new_spec*)
Add a new species to the first empty place and return the position of the lineage.

Return the location the species has been added to.

Parameters

- `new_spec`: the new species reference to place in the first empty space.

void **addSpeciesSilent**(unsigned long *new_spec*)
Add a new species to the first empty place. Essentially a version of *addSpecies()* without returning the species location.

Parameters

- `new_spec`: the new species reference to place in the first empty space.

void **deleteSpecies**(int *index*)
Removes the species at the specified index. The species number will be replaced with 0, indicating no species present.

Older versions of this function re-shuffled the list so that all species came at the top.

Parameters

- `index`: the index of the species to remove from the list.

void **decreaseNwrap**()
Decreases the nwrap by one.

Indicates the number of species wrapped at the location of this *SpeciesList* object has decreased by one.

void **increaseNwrap**()
Increases the nwrap by one.

Indicates the number of species wrapped at the location of this *SpeciesList* object has increased by one.

void **changePercentCover**(unsigned short *newmaxsize*)
Changes the maximum size of the *SpeciesList*. Creates a new list object with all the species in the correct place from the old list object and zeros everywhere else.

Parameters

- `newmaxsize`: the new maximum size to be applied.

unsigned long **getRandLineage**(*NRrand* &*rand_no*)
Get a random species reference number from all the potential entries. Updated alternative version returns any entry, including empty cells, giving the probability of coalescence as well.

Return the reference of the random lineage. 0 indicates an empty space.

Parameters

- `rand_no`: the random number object to pass (for maintaining the same seed throughout simulations).

unsigned long **getSpecies**(unsigned short *index*)
Get the species reference number from a particular entry.

Return the species reference at the specified location.

Parameters

- `index`: the location of the species to reference.

unsigned long **getNext**()
Get the next_active variable.

Return the next linked species reference.

unsigned short **getNwrap**()
Getter for the nwrap.

Return the number of wrapped lineages currently at this grid cell.

int **getListsize**()
Getter for the list size.

Return the number of lineages currently directly within the *SpeciesList*.

int **getMaxsize**()
Getter for the maximum size of the *SpeciesList* object.

Return the maximum number of lineages that can exist currently.

void **wipeList**()
Empties the list of any data and fills the list with zeros.

Friends

ostream &**operator<<**(ostream &*os*, **const** *SpeciesList* &*r*)
Outputs the *SpeciesList* object to an output stream. Allows for piping to the terminal or writing the object to a file.

Return the output stream.

Parameters

- `os`: the output stream.
- `r`: the *SpeciesList* object to output.

istream &**operator>>**(istream &*is*, *SpeciesList* &*r*)
Inputs the *SpeciesList* object from an input stream. Allows for reading data from a file or string stream.

Parameters

- `is`: the input stream.
- `r`: the *SpeciesList* object to input to.

Class Tree

- Defined in *File Tree.h*

class `Tree`

Represents the output phylogenetic tree.

Contains all functions for running simulations, outputting data and calculating coalescence tree structure.

Public Functions

`Tree`()

The constructor for the tree object.

Sets all uninitiated variables to false, except log_all. log_all should be changed to false if minimal text output during simulations is desired.

void **`setMapVars`**(vector<string> *comargs*, bool *fullmode*)

Sets the map object with the correct variables, taking the *Mapvars* structure defined elsewhere for the parameters.

This function can only be run once, otherwise a *Main_Exception* will be thrown

Parameters

- `comargs`: a vector of command-line arguments to be parsed
- `fullmode`: if true, the simulation is using full command-line flags

vector<double> **`getTemporalSampling`**()

Gets the map autocorrel times.

void **`importMaps`**()

Imports the maps into the forestmap object.

The simulation variables should have already been imported by *setMapVars()*, otherwise a *Fatal_Exception* will be thrown.

long long **`getSeed`**()

Getter for the simulation seed.

Return Returns the seeds

void **`setSeed`**(long long *theseedin*)

Sets the simulation seed to the provided variable.

The seed is set within the NR object. This will be fixed for the simulation and is only performed once.

Parameters

- `theseedin`: the desired seed to set for the simulation

vector<long> **`randomList`**(long *maxnum*, long *numnum*)

Deprecated function which generated a random list.

Return

Parameters

- `maxnum`:
- `numnum`:

void **`setup`**(vector<string> &*comargs*, bool *fullmode*)
The setup function for parsing the command line arguments and generating the simulation objects.

The simulation parameters are set from comargs using *setMapVars()*. Generates and fills the active and grid objects as well as importing all the maps from the supplied files. *importMaps()* is called for importing the map files

Parameters

- `comargs`: a vector of the command-line arguments which are parsed using *setMapVars()*.
- `fullmode`: if true, the command line options are fully specified, rather than in the updated condensed form

void **`removeOldPos`**(**const** unsigned long &*chosen*)
Removes the old position within active by checking any wrapping and removing connections.

The function also corrects the linked list to identify the correct nwrap for every wrapped lineage in that space.

Parameters

- `chosen`: the desired active reference to remove from the grid.

void **`calcMove`**(long &*oldx*, long &*oldy*, long &*oldxwrap*, long &*oldywrap*)
Calculate the move, given a start x,y coordinates and wrapping.

The provided parameters will be altered to contain the new values so no record of the old variables remains after function running. Current dispersal methods use a fattailed dispersal.

Parameters

- `oldx`: the old x position on the grid
- `oldy`: the old y position on the grid
- `oldxwrap`: the old x wrapping
- `oldywrap`: the old y wrapping

long double **`calcMinMax`**(**const** unsigned long &*current*)
Calculates the minmax for a given branch.

Calculates the speciation rate required for speciation to have occured on this branch.

Parameters

- `current`: the current active reference to perform calculations over.

void **`coalescenceEvent`**(**const** unsigned long &*chosen*, unsigned long &*coalchosen*)
Perform the coalescence between lineages. Once coalesced, lineages are removed from the active scope.

Parameters

- `chosen`: the chosen lineage for coalescence
- `coalchosen`: the target lineage for coalscence

void **`calcNewPos`**(bool &*coal*, **const** unsigned long &*chosen*, unsigned long &*coalchosen*, **const** long &*oldx*, **const** long &*oldy*, **const** long &*oldxwrap*, **const** long &*oldywrap*)

Calculates the new position, checking whether coalescence has occured and with which lineage.

This involves correct handling of checking wrapped lineages (outside the original grid). The probability of coalescence is also calculated.

Parameters

- `coal`: boolean for whether coalescence occured or not
- `chosen`: the chosen lineage
- `coalchosen`: the lineaged that is coalescing (if required)
- `oldx`: the old x position
- `oldy`: the old y position
- `oldxwrap`: the old x wrapping
- `oldywrap`: the old y wrapping

void **`switchPositions`**(**const** unsigned long *chosen*)

Switches the chosen position with the endactive position.

Parameters

- `chosen`: the chosen lineage to switch with endactive.

void **`speciation`**(**const** unsigned long &*chosen*)

Speciation to supplied lineage.

Also calls the *removeOldPos()* and *switchPositions()* functions for removing the lineage out of active reference.

Parameters

- `chosen`:

unsigned long **`estSpecnum`**()

Estimates the species number from the second largest minimum speciation rate remaining in active.

This allows for halting of the simulation once this threshold has been reached. However, the function is not currently in use as calculating the coalescence tree is very computionally intensive.

void **`runChecks`**(**const** unsigned long &*chosen*, **const** unsigned long &*coalchosen*)

Run checks at the end of each cycle which make certain the move has been successful.

Parameters

- `chosen`: the chosen lineage to check
- `coalchosen`: the lineage which is coalescing with the chosen lineage which we are also required to check

void **validationCheck**(**const** unsigned long &*chosen*, int *o*, **const** unsigned long &*current*, **const** bool &*coal*)
Validates the chosen lineage is set up correctly.

This functions is for debugging purposes. It checks that the wrapping of the lineage in active is correct, and links to other lineages are applied correctly.

Parameters

- `chosen`: the desired lineage to check
- `o`: an integer for outputting and referencing later on
- `current`: the current lineage in memory, again for referencing purposes.
- `coal`: boolean for whether coalescence has occured.

void **checkSimSize**(unsigned long *req_data*, unsigned long *req_active*)
Checks the size of the main active and data objects is large enough.

Parameters

- `req_data`: the required data object size
- `req_active`: the required active object size

bool **runSim**()
Run the entire simulation given the start conditions already defined by *setup()*

Setup is assumed to have been run already. This function is the main function containing the main loop of the simulation. At the end of the simulation, returns true if the simulation is complete, false otherwise.

void **expandMap**(double *generationin*)
Expands the map, generating the new lineages where necessary.

The samplemask provided is used for expansion. Any empty spaces are filled with a new lineage. Lineages which have not moved are changed to tips, with a new data entry so that original and new generations are recorded.

Parameters

- `generationin`: the generation that the expansion is occuring at. This is used in recording the new tips.

unsigned long **sortData**()
Sort and process the species list so that the useful information can be extracted from it.

void **outputData**()
Outputs important simulation data to a csv file. Overloaded version which automatically calls *sortData()* if no species richness is provided.

void **outputData**(unsigned long *species_richness*)
Outputs important simulation data to a csv file. This function will likely be remove in future versions as all simulation output is now contained in an SQLite database.

Parameters

- `species_richness`: the species richness of the tree with the minimum speciation rate. Outputted into the csv file.

void **writeTimes**()
Writes the times to the terminal for simulation information.

void **simPause**()
Pause the simulation and dump data from memory.

void **simResume**(bool *cont_sim*)
Resumes the simulation from a previous state.

Reads in the parameters and objects from file and re-starts the simulation.

Parameters

- `cont_sim`: whether or not to resume simulations, imported from a call to checkSims().

void **sqlCreate**()
Generates the SQL database file from the full simulation data. This allows for greater analysis of the data after completion of the simulation.

void **sqlOutput**()
Copy the in-memory database to file.

This function should not be called if the database is already opened on disc, and won't do anything if it is.

void **applySpecRates**(double *sr*, double *t*)
Applies the given speciation rate to the tree.

Parameters

- `sr`: the required speciation rate.

void **applySpecRates**(double *sr*)
Overloaded version of applySpecRates for the default generation (0.0).

Parameters

- `sr`: the speciation rate to apply to the tree.

Class Treelist

- Defined in *File Treelist.h*

class Treelist
A class to contain the tree object lineages and reconstructing the coalescence tree. Contains functions for calculating the number of species for a given speciation rate, outputting spatial data and generating species abundance distributions. Requires a link to the SQLite database from simulation output, and produces results within the same database file.

Public Functions

Treelist(*Row*<*Treenode*> &*r*)
Contructor for treelist.

Parameters

- `r`: *Row* of *Treenode* objects to link to.

void **setList**(*Row*<*Treenode*> &*l*)
Set the nodes object to the input *Row* of *Treenode* objects.

Parameters

- `l`: the *Row* of *Treenode* objects to link to.

void **setDatabase**(sqlite3 **dbin*)
Sets the database object for the sqlite functions.

Parameters

- `dbin`: the sqlite3 input database.

bool **hasSetMem**()
Get the boolean of whether the memory has been set yet.

Return the memory boolean.

double **getMinimumSpeciation**()
Get the minimum speciation rate the simulation was originally run with. This value is read in from the SIMULATION_PARAMETERS table in the database file.

Return the minimum speciation rate.

void **importSamplemask**(string *sSamplemask*)
Imports the samplemask if it hasn't already been imported.

Parameters

- `sSamplemask`: the path to the samplemask file.

unsigned long **countSpecies**()
Counts the number of species that have speciated currently on the tree.

Return the number of species

bool **doubleCompare**(double *d1*, double *d2*, double *epsilon*)
Compares two doubles and returns a boolean of whether they are equal, within the epsilon deviation. This is useful for floating point errors in saving and reading doubles from file.

Return

Parameters

- `d1`: the first double.
- `d2`: the second double.
- `epsilon`: the deviation within which the values are assumed to be equal.

unsigned long **calcSpecies**(double *s*)
Calculate the number of species in the list given a specific speciation rate. This is the main function which reconstructs the coalescence tree. Each *Treenode* object will end having its existance value set correctly after a call to this function.

Return the number of species present.

Parameters

- `s`: the speciation rate to apply.

void **`calcSpeciesAbundance`**()
Calculates the species abundance of the dataset. The species abundances will be with rOut after a call do this function. If a samplemask has been applied, only lineages which originally existed in the samplemask will be counted.

void **`resetTree`**()
Resets the entire tree. Sets existance to false, speciation to false and removes any species ID.

void **`detectDimensions`**(string *db*)
This function detects the maximum x and y values of the sql database. This allows for the dimensions before opening the map file.

Parameters

- `db`: the path to the input database to read from.

void **`importData`**(string *inputfile*)
Imports the data from the desired SQL database object into the array. It also stores an in-memory version of the data for working with.

Parameters

- `inputfile`: the path to the input SQLite database.

void **`setGeneration`**(long double *generationin*)
Set the time of interest where all analysis will be performed. It is assumed that this was one of the temporal points defined at simulation time.

Parameters

- `generationin`: the target generation.

void **`createDatabase`**(double *s*)
Creates a new table in the database file and outputs the database object to the same file as the input file. The new SPECIES_ABUNDANCES table contains the species abundance distribution for the whole samplemask. A similar tabe FRAGMENT_ABUNDANCES is generated by *createFragmentDatabase()* if specified via the command line parameters.

Parameters

- `s`: the speciation rate to apply.

void **`createFragmentDatabase`**(**const** *Fragment* &*f*)
Creates a new table in the database file and outputs the database object to the same file as the input file. Essentially creates a species abundance distribution (as in *createDatabase()*), but for the specified fragment within the samplemask.

Parameters

- `f`: the *Fragment* to sample from.

void **exportDatabase**(string *outputfile*)
Output the database from memory to the database file. Most of the time, it is desirable for the outputfile to be the same path as the input file and will write to the same object.

Parameters

- `outputfile`: the path to the output file.

void **recordSpatial**()
Record the full spatial data. Creates a new table, SPECIES_LOCATIONS containing every species and their parameters. This allows for more in-depth analysis to be performed if necessary.

void **calcFragments**(string *fragment_file*)
Calculates the limits of each fragment in the sample map and adds it to the vector of fragments. If the fragment_file is null, then the program will attempt to calculate fragments from the map.

Parameters

- `fragment_file`: the fragment file to read from.

void **applyFragments**()
Calculate species abundances for each fragment, and call *createFragmentDatabase()* for each *Fragment*.

Class Treenode

- Defined in *File Treenode.h*

class Treenode
The *Treenode* class that acts as a data storage object for the phylogenetic tree.

Also contains all the necessary routines for changes to a lineage's attributes, called by *Treelist* objects when generating new coalescence trees.

Public Functions

Treenode()
The default constructor.

~Treenode()
The default destructor.

void **setup**(bool *z*, unsigned long *xp*, unsigned long *yp*, long *xi*, long *yi*)
Sets up variables with initial conditions.

Parameters

- `z`: whether this lineage is a tip or not (represents the end of a tree).
- `xp`: the x position on the grid.
- `yp`: the y position on the grid.
- `xi`: the number of wraps in the x dimension.
- `yi`: the number of wraps in the y dimension.

void **setup**(bool *z*, unsigned long *xp*, unsigned long *yp*, long *xi*, long *yi*, long double *generation*)
Overloaded *setup()* function, additionally taking a generation time point.

Used when creating lineages after the start of the simulation, when the generation is not 0.

Parameters

- `z`: whether this lineage is a tip or not (represents the end of a tree).
- `xp`: the x position on the grid.
- `yp`: the y position on the grid.
- `xi`: the number of wraps in the x dimension.
- `yi`: the number of wraps in the y dimension.
- `generation`: the current generation at creation time.

void **setExistance**(bool *b*)
Setter for the existance of the lineage.

Parameters

- `b`: existance boolean.

void **setParent**(unsigned long *x*)
Setter for the parent reference.

Parameters

- `x`: a reference for the parent location.

void **qReset**()
Resets the lineage. Remove any species ID, existance and speciation record.

void **setPosition**(long *x*, long *y*, long *xw*, long *yw*)
Set a new position for the lineage.

Parameters

- `x`: the x position on the grid.
- `y`: the y position on the grid.
- `xw`: the number of wraps in the x dimension.
- `yw`: the number of wraps in the y dimension.

void **setSpec**(long double *d*)
Setter for the randomly generated number (from NRrand.d0()) for the speciation probability for this lineage.

Parameters

- `d`: the speciation probability.

void **setIGen**(unsigned long *g*)
Setter for the number of generations this lineage has existed for.

Parameters

- g: the number of generations that the lineage has existed.

void **setGeneration**(long double *d*)
Setter for the birth generation timer for the lineage.

Note that moves that don't involve coalescence do not create a new *Treenode* object, and therefore the generation_added does not get updated. However, coalescence events will cause a new *Treenode* object creation. The lineage birth generation is generally only important for calculating the age of tips.

Parameters

- d:

void **setSpeciation**(bool *s*)
Setter for the speciation boolean.

Parameters

- s: the speciation boolean.

void **burnSpecies**(unsigned long *idin*)
Setter for the species ID. Once set to something other than 0, this cannot be changed with a call to *qReset()* or *resetSpecies()*.

Parameters

- idin: the species ID.

void **setTip**(bool *b*)
Setter for the tip boolean.

Parameters

- b: the tip boolean.

void **resetSpecies**()
Reset the species ID to 0.

void **increaseGen**()
Increases the generation counter by one.

bool **getExistance**()
Getter for the existance boolean.

Return the existance boolean.

bool **isTip**()
Getter for the tip boolean.

Return the tip boolean.

unsigned long **getParent**()
Getter for the parent location.

Return the parent reference.

unsigned long **getXpos**()
Getter for the x position.

Return the x position.

unsigned long **getYpos**()
Getter for the y position.

Return the y position.

long **getXwrap**()
Getter for the number of times the lineage is wrapped in the x dimension.

Return the number of times the lineage is wrapped in the x dimension.

long **getYwrap**()
Getter for the number of times the lineage is wrapped in the y dimension.

Return the number of times the lineage is wrapped in the y dimension.

bool **hasSpeciated**()
Getter for the speciation boolean.

Return the speciation boolean.

unsigned long **getSpeciesID**()
Getter for the species ID.

Return the species ID.

long double **getSpecRate**()
Getter for the randomly generated speciation probability.

Return the speciation probability.

unsigned long **getGenRate**()
Getter for the number of generations the lineage has existed.

Return the number of generations of existance.

long double **getGeneration**()
Getter for the generation the lineage was created.

Return the generation counter the lineage was created.

void **speciate**()
Sets the speciation boolean to true.

Friends

ostream &**operator<<**(ostream &*os*, **const** *Treenode* &*t*)
Overloading the << operator for outputting a *Treenode* object to an output stream.

Return the output stream.

Parameters

- `os`: the output stream.
- `t`: a *Treenode* object to output.

istream &**operator>>**(istream &*is*, *Treenode* &*t*)
Overloading the >> operator for inputting the *Treenode* object from an input stream.

Return the input stream.

Parameters

- `is`: the input stream.
- `t`: a *Treenode* object to input to.

Functions

Function charconvertor

- Defined in *File Setup.cpp*

long **charconvertor**(char *charin*)
converts the characters to an integer for command-line parsing.

Originally written for handy HPC use.

Return the converted integer.

Parameters

- `charin`: the character to change to an integer.

Function checkFolders

- Defined in *File Setup.cpp*

void **checkFolders**(string *sFineMap*, string *sCoarseMap*, string *sFineMapPristine*, string *sCoarseMapPristine*, string *sOutputFolder*, string *sSampleMask*)
Runs doesExist() on every supplied map, providing defaults where possible if one doesn't.

Parameters

- `sFineMap`: the fine map file path
- `sCoarseMap`: the course map file path
- `sFineMapPristine`: the fine pristine map file path
- `sCoarseMapPristine`: the coarse pristine map file path
- `sOutputFolder`: the output folder

- `sSampleMask`: the sample mask file path

Function checkSims

- Defined in *File Setup.cpp*

bool **`checkSims`**(int *the_task*, int *the_seed*, string *outdirect*)
Checks for previous simulations and resumes simulations if required.

Return returns whether or not there is a file to be resumed from.

Parameters

- `the_task`: the task reference for reading the files
- `the_seed`: the seed reference
- `outdirect`: the directory that the files were originally outputted to.

Function checkSpeciation

- Defined in *File Treelist.cpp*

bool **`checkSpeciation`**(long double *random_number*, long double *speciation_rate*, int *number_of_generations*)
Checks whether speciation has occured for the provided parameters. Provided here for ease of use when bug-fixing.

Return bool the speciation state of the lineage

Parameters

- `random_number`: the random number associated with a lineage
- `speciation_rate`: the global speciation rate
- `number_of_generations`: the number of generations the lineage has existed

Function doesExist

- Defined in *File Setup.cpp*

bool **`doesExist`**(string *testfile*)
Checks the existance of a file on the hard drive.

Return if true, file exists

Parameters

- `testfile`: the file to examine

Function doesExistNull

- Defined in *File Setup.cpp*

bool **doesExistNull**(string *testfile*)
Checks for the existance of a file, but returns true if the file name is 'null'. Note: this function just calls doesExist().

Return if true, file exists (or is null).

Parameters

- `testfile`: the file to examine

Function getSR

- Defined in *File Setup.cpp*

void **getSR**(int *argc*, **const** vector<string> &*comargs*, *Tree* &*t*)
Applies the speciation rates provided on the coalescence tree post-simulation.

Parameters

- `argc`: the number of arguments initially provided
- `comargs`: a vector of command line arguments to parse.
- `t`: a *Tree* object containing the coalescence tree from a completed simulation.

Function importArgs

- Defined in *File Config.cpp*

void **importArgs**(**const** unsigned int &*argc*, char **argv*[], vector<string> &*comargs*)
Import the command line arguments in to the vector for future processing. Arguments will be placed in the vector comargs.

Parameters

- `argc`: the number of arguments.
- `argv`: a point to the array of arguments in raw character form.
- `comargs`: a vector of the command-line arguments to be filled.

Function importConfig

- Defined in *File Setup.cpp*

int `ConfigOption::`**importConfig**(vector<string> &*comargs*)
Imports the parameters from the config file and returns an integer of the number of arguments.

Return a count of the number of arguments (should also be the size of comargs).

Parameters

- `comargs`: a vector of command line arguments to import to from file.

Function jobConvertor

- Defined in *File Setup.cpp*

long **jobConvertor**(char **argin*)
Converts the job character into a job number.

Return the job integer.

Parameters

- `argin`: the character to convert

Function main

- Defined in *File main.cpp*

Warning: doxygenfunction: Cannot find function "main" in doxygen xml output for project "NECSim" from directory: ../PyCoalescence/NECSim/Documentation/xml/

Function openLogFile

- Defined in *File Setup.cpp*

void **openLogFile**(bool *append*)
Opens the log file for redirecting stdout.

Parameters

- `append`: if true, appends to the existing log file.

Function parseArgs

- Defined in *File Setup.cpp*

bool **parseArgs**(int &*argc*, vector<string> &*comargs*)
Parse the command line parameters, taking in to account all the options for default runs, showing help options and importing from a config file.

This version of the functions operates when infinite landscapes are not in order,

Return boolean set to true if the arguments are to be parsed in the new method. False for old, verbose method.

Parameters

- `argc`: the number of arguments initially provided
- `comargs`: a vector of command line arguments to parse.

Function removeComOption

- Defined in *File Setup.cpp*

void **removeComOption**(int &*argc*, vector<string> &*comargs*)
Removes the command line options supplied, leaving just a clean vector with the correct data in.

Function runAsDefault

- Defined in *File Setup.cpp*

void **runAsDefault**(vector<string> &*comargs*)
Sets up the command-line arguments for default parameters.

This is intended for testing purposes only.

Parameters

- `comargs`: a vector of command-line arguments for putting the parameters into.

Function runLarge

- Defined in *File Setup.cpp*

void **runLarge**(vector<string> &*comargs*)
Sets up the command-line arguments for larger-scale default parameters.

This is intended for testing purposes only.

Parameters

- `comargs`: a vector of command-line arguments for putting the parameters into.

Function runXL

- Defined in *File Setup.cpp*

void **runXL**(vector<string> &*comargs*)
Sets up the command-line arguments for default very large scale parameters.

This is intended for testing purposes only.

Parameters

- `comargs`: a vector of command-line arguments for putting the parameters into.

Variables

Variable gdal_data_sizes

- Defined in *File Matrix.h*

const int **gdal_data_sizes**[] = {0,8,16,16,32,32,32,64}

Variable log_name

- Defined in *File Setup.cpp*

string **log_name**

Variable saved_stdout

- Defined in *File Setup.cpp*

int **saved_stdout**

Defines

Define AM

- Defined in *File Fattaildeviate.h*

AM

Define EPS

- Defined in *File Fattaildeviate.h*

EPS

Define IA1

- Defined in *File Fattaildeviate.h*

IA1

Define IA2

- Defined in *File Fattaildeviate.h*

IA2

Define IM1

- Defined in *File Fattaildeviate.h*

IM1

Define IM2

- Defined in *File Fattaildeviate.h*

IM2

Define IMM1

- Defined in *File Fattaildeviate.h*

```
IMM1
```

Define IQ1

- Defined in *File Fattaildeviate.h*

```
IQ1
```

Define IQ2

- Defined in *File Fattaildeviate.h*

```
IQ2
```

Define IR1

- Defined in *File Fattaildeviate.h*

```
IR1
```

Define IR2

- Defined in *File Fattaildeviate.h*

```
IR2
```

Define MATRIX

- Defined in *File Matrix.h*

```
MATRIX
```

Define NDIV

- Defined in *File Fattaildeviate.h*

```
NDIV
```

Define NTAB

- Defined in *File Fattaildeviate.h*

```
NTAB
```

Define null

- Defined in *File Matrix.h*

```
null
```

Define RNMX

- Defined in *File Fattaildeviate.h*

```
RNMX
```

Define sql_ram

- Defined in *File Tree.h*

```
sql_ram
```

Macro for using the fast-cpp-csv-parser from Ben Strasser (available from https://github.com/ben-strasser/fast-cpp-csv-parser). This enables much faster csv reading, but can cause problems on systems where this module is not fully tested.

Macro to compile using RAM for storage of the active SQL database. Without this, the database will be written directly to disc (which is slower, but an option if RAM requirements get too huge). For HPC systems, it is recommended to use this option as write speeds are generally fast and large simulations don't have a linear increase in the SQL database size (at least in RAM).

Define version1_01

- Defined in *File Datapoint.h*

```
version1_01
```

Define version1_11

- Defined in *File Matrix.h*

```
version1_11
```

Files

File Config.cpp

Definition (`Config.cpp`)

Program Listing for File Config.cpp

- Return to documentation for *File Config.cpp*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for
↪full license details.
//
#include "Config.h"
void importArgs(const unsigned int& argc, char* argv [], vector<string>& comargs)
{
    for(unsigned int i = 0; i < argc; i++)
    {
        comargs.push_back(string(argv[i]));
    }
    // check size is correct
    if(comargs.size() != argc)
    {
        cerr << "ERROR_MAIN_010: Incorrect command line parsing." << endl;
    }
}
```

Includes

- `Config.h` (*File Config.h*)

Functions

- *Function importArgs*

File Config.h

Definition (`Config.h`)

Program Listing for File Config.h

- Return to documentation for *File Config.h*

```
// This file is part of NECSim project which is released under BSD-3 license.
// See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause)
↪for full license details.
//
// Header guard
//
#ifndef CONFIGCLASS
#define CONFIGCLASS

/*************************************************************

↪                                                             ↪
↪                        INCLUDES
 *************************************************************/
#include <string>
#include <iostream>
#include <sstream>
#include <fstream>
```

```
#include <stdexcept>
#include <vector>
#include <cstring>
#include <unistd.h>
#include <cmath>
#include <cctype>
#include <algorithm>
#include "CustomExceptions.h"
using namespace std;
using std::string;
void importArgs(const unsigned int& argc, char* argv[], vector<string>& comargs);

//

struct SectionOption
{
    string section;
    vector<string> val;
    vector<string> refs;

    string getOption(string refval)
    {
    for(unsigned int i = 0; i < refs.size(); i++)
        {
        if(refs[i] == refval)
            {
            return (val[i]);
            }
        }
#ifdef DEBUG
    cerr << "Reference not found in keyoption." << endl;
#endif
    return ("null");
    }

    friend ostream& operator<<(ostream& os, const SectionOption& k)
    {
    os << k.section << "\n" << k.val.size() << "\n" << k.refs.size() << "\n";
    for(unsigned int i = 0; i < k.val.size(); i++)
        {
        os << k.val[i] << "\n";
        }
    for(unsigned int i = 0; i < k.refs.size(); i++)
        {
        os << k.refs[i] << "\n";
        }
    return os;
    }

    friend istream& operator>>(istream& is, SectionOption& k)
    {
    // os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
    unsigned int valsize, refsize;
    is >> k.section >> valsize >> refsize;
    string tmp;
    for(unsigned int i = 0; i < valsize; i++)
        {
        is >> tmp;
```

```
        k.val.push_back(tmp);
        }
    for(unsigned int i = 0; i < refsize; i++)
        {
        is >> tmp;
        k.refs.push_back(tmp);
        }
    return is;
    }
};

class ConfigOption
{
   private:
    string configfile;
    bool bConfig;
    bool bMain;  // is true if this is the main command line import (and therefore we
↪want to delete the first few
    // command line options)
    bool bFullParse;  // if this is true, each KeyOption structure will be returned
↪after each read.
    vector<SectionOption> configs;  // all config data if full parse is true.
   public:
    ConfigOption()
    {
    bConfig = false;
    configfile = "none";
    }

    string getFile()
    {
        return configfile;
    }

    bool getMain()
    {
        return bMain;
    }

    bool getFull()
    {
        return bFullParse;
    }

    void setConfig(string file, bool main, bool full_parse = false)
    {
        if(!bConfig)
        {
            bMain = main;
            configfile = file;
            bConfig = true;
            bFullParse = full_parse;
        }
        else
        {
            throw Config_Exception("Attempt to set config file twice.");
        }
    }
```

```
    void parseConfig()
    {
        ifstream is_file;
        try
        {
            is_file.open(configfile);
        }
        catch(...)
        {
            throw Config_Exception(
                "ERROR_CONF_004c: Could not open the config file. Check file exists␣
↪and is readable.");
        }
        if(!is_file.fail())
        {
            string line;
            // Get the first line of the file.
            while(getline(is_file, line))
            {
//              cout << line << endl;
                istringstream is_line(line);
                string key;
                string val;
                // Skip all whitespace
                is_line >> skipws;
                // start a new section
                if(line[0] == '[')
                {
                    SectionOption tempSections;
                    // get the section name
                    string section;
                    if(getline(is_line, section, ']'))
                    {
                        section = section.erase(0, 1);
                        tempSections.section = section;
//                      cout << section << endl;
                    }
                    // read each line
                    while(getline(is_file, line))
                    {
                        // end the section when a new one starts.
                        if(line[0] == '[' || line.size() == 0)
                        {
                            break;
                        }
                        istringstream is_line2(line); // update the input-line stream
                        if(getline(is_line2, key, '='))
                        {
                            key.erase(std::remove(key.begin(), key.end(), ' '), key.
↪end());

                            is_line2 >> skipws;
                        }
                        if(!is_line2)
                        {
//                          cout << is_line2 << endl;
                            throw Config_Exception("ERROR_CONF_001: Read error in␣
↪config file.");
```

```
                        }
                        if(getline(is_line2, val))
                        {
                            val.erase(std::remove(val.begin(), val.end(), ' '), val.
↪end());
                        }
                        if(!is_line2)
                        {
                            throw Config_Exception("ERROR_CONF_001: Read error in␣
↪config file.");
                        }
                        tempSections.refs.push_back(key);
                        tempSections.val.push_back(val);
                    }
                    configs.push_back(tempSections);
                }
            }
        }
        else
        {
            throw Config_Exception(
                "ERROR_CONF_004b: Could not open the config file. Check file exists␣
↪and is readable.");
        }
        if(is_file.eof())
        {
            is_file.close();
        }
        else
        {
            throw Config_Exception("ERROR_CONF_002: End of file not reached. Check␣
↪input file formating.");
        }
    }

    void testPrint()
    {
        cout << configs.size() << endl;
        for(unsigned int i = 0; i < configs.size(); i++)
        {
            cout << configs[i].section << endl;
            for(unsigned j = 0; j < configs[i].refs.size(); j++)
            {
                cout << configs[i].refs[j] << " = " << configs[i].val[j] << " , ";
            }
            cout << endl;
        }
    }
    vector<SectionOption> getSectionOptions()
    {
        return configs;
    }

    SectionOption operator[](int index)
    {
        return (configs[index]);
    }
    unsigned int getSectionOptionsSize()
```

```
    {
        return (configs.size());
    }

    vector<string> getSections()
    {
        vector<string> toret;
        for(unsigned i = 0; i < configs.size(); i ++)
        {
            toret.push_back(configs[i].section);
        }
        return(toret);
    }

    bool hasSection(string sec)
    {
        for(unsigned i = 0; i < configs.size(); i++)
        {
            if(configs[i].section == sec)
            {
                return(true);
            }
        }
        return(false);
    }

    vector<string> getSectionValues(string sec)
    {
        for(unsigned i = 0; i < configs.size(); i++)
        {
            if(configs[i].section == sec)
            {
                return(configs[i].val);
            }
        }
        throw Config_Exception("Section not found in config file: " + sec);
    }
    string getSectionOptions(string section, string ref)
    {
        for(unsigned int i = 0; i < configs.size(); i++)
        {
            if(configs[i].section == section)
            {
                for(unsigned int j = 0; j < configs[i].refs.size(); j++)
                {
                    if(configs[i].refs[j] == ref)
                    {
                        return (configs[i].val[j]);
                    }
                }
            }
        }
#ifdef DEBUG
        cerr << "No reference found for " << section << ", " << ref << endl;
#endif
        return ("null");
    }
    int importConfig(vector<string>& comargs)
```

```
    {
    // Check that the previous arguments have already been imported.
        if(bMain)
        {
            if(comargs.size() != 3)
            {
                throw Config_Exception(
                    "ERROR_CONF_003: Number of command line arguments not correct␣
↪before import.");
            }
        }
        ifstream is_file;
        try
        {
            is_file.open(configfile);
        }
        catch(...)
        {
            throw Config_Exception(
                "ERROR_CONF_004a: Could not open the config file. Check file exists␣
↪and is readable.");
        }
        if(!is_file.fail())
        {
            string line;
            while(getline(is_file, line))
            {
                istringstream is_line(line);
                string key;
                is_line >> skipws;
                if(line[0] == '[')
                {
                    continue;
                }
                if(getline(is_line, key, '='))
                {
                    // Could implement proper data parsing based on the key object.
                    is_line >> skipws;
                    string value;
                    if(getline(is_line, value))
                    {
                        value.erase(std::remove(value.begin(), value.end(), ' '),␣
↪value.end());
                        if(!is_line)
                        {
                            cout << value << endl;
                            throw Config_Exception("ERROR_CONF_001: Read error in␣
↪config file.");
                        }
                        char* tmp = new char[value.length() + 1];
                        strcpy(tmp, value.c_str());
                        comargs.push_back(tmp);
                    }
                }
            }
        }
        else
        {
```

```
            throw Config_Exception(
                "ERROR_CONF_004b: Could not open the config file. Check file exists
↪and is readable.");
        }
        if(is_file.eof())
        {
            is_file.close();
        }
        else
        {
            throw Config_Exception("ERROR_CONF_002: End of file not reached. Check
↪input file formating.");
        }
        if(bMain)
        {
            // remove the file name from the command line arguments to maintain the
↪vector format.
            comargs.erase(comargs.begin() + 2);
        }
        return comargs.size();
    }

    friend ostream& operator<<(ostream& os, const ConfigOption& c)
    {
        os << c.configfile << "\n" << c.bConfig << "\n" << c.bMain << "\n" << c.
↪bFullParse << "\n" << c.configs.size()
           << "\n";
        for(unsigned int i = 0; i < c.configs.size(); i++)
        {
            os << c.configs[i];
        }
        return os;
    }

    friend istream& operator>>(istream& is, ConfigOption& c)
    {
        // os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        unsigned int configsize;
        is >> c.configfile >> c.bConfig >> c.bMain >> c.bFullParse >> configsize;
        SectionOption tmpoption;
        // cout << "configsize: " << configsize << endl;
        if(configsize > 0)
        {
            for(unsigned int i = 0; i < configsize; i++)
            {
                is >> tmpoption;
                c.configs.push_back(tmpoption);
            }
        }
        // cout << "end config" << endl;
        return is;
    }
};
#endif
```

Includes

- `CustomExceptions.h` (*File CustomExceptions.h*)
- `algorithm`
- `cctype`
- `cmath`
- `cstring`
- `fstream`
- `iostream`
- `sstream`
- `stdexcept`
- `string`
- `unistd.h`
- `vector`

Included By

- *File Config.cpp*
- *File Map.h*
- *File Tree.h*
- *File SpeciationCounter.cpp*

Namespaces

- *Namespace std*

Classes

- *Struct SectionOption*
- *Class ConfigOption*

Functions

- *Function importArgs*

File CustomExceptions.h

Definition (`CustomExceptions.h`)

Program Listing for File CustomExceptions.h

- Return to documentation for *File CustomExceptions.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for full license details.
// Author: Samuel Thompson
// Contact: samuel.thompson14@imperial.ac.uk or thompsonsed@gmail.com
 #ifndef EXCEPTION
 #define EXCEPTION
 #include <stdexcept>
 using namespace std;
struct Main_Exception : public runtime_error
{
  Main_Exception():runtime_error("Exception thrown at run time in main: "){}
  Main_Exception(string msg):runtime_error(msg.c_str()){}
};


struct Fatal_Exception : public runtime_error
{
  Fatal_Exception():runtime_error("Fatal exception thrown at run time, quitting program. "){}
  Fatal_Exception(string msg):runtime_error(msg.c_str()){}
};

 // for non fatal map exceptions - in most cases, if they can't be resolved, the program will have to quit regardless.
 struct Map_Exception : public runtime_error
{
    Map_Exception():runtime_error("Exception thrown at run time in Map: "){}
    Map_Exception(string msg):runtime_error(msg.c_str()){}
};

 // For returning fatal map exceptions
 struct Map_Fatal_Exception : public runtime_error
 {
    Map_Fatal_Exception():runtime_error("Fatal exception thrown at run time in Map: "){}
    Map_Fatal_Exception(string msg):runtime_error(msg.c_str()){}
 };

 /*********************************************************
                        CONFIGOPTION CLASS
 *********************************************************/
 // Create the ConfigException object
struct Config_Exception : public runtime_error
{
  Config_Exception():runtime_error("Exception thrown at run time in config: "){}
  Config_Exception(string msg):runtime_error(msg.c_str()){}
};
```

```
using std::runtime_error;
struct SpeciesException : public runtime_error
{
    SpeciesException():runtime_error("Exception thrown at run time in␣
↪SpeciationCounter: "){}
    SpeciesException(string msg):runtime_error(msg.c_str()){}
};
#endif
```

Includes

- stdexcept

Included By

- *File Config.h*
- *File Treelist.h*

Classes

- *Struct Config_Exception*
- *Struct Fatal_Exception*
- *Struct Main_Exception*
- *Struct Map_Exception*
- *Struct Map_Fatal_Exception*
- *Struct SpeciesException*

File Datapoint.h

Definition (`Datapoint.h`)

Program Listing for File Datapoint.h

- Return to documentation for *File Datapoint.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
// Datapoint.cpp version 3.1
// Author - Samuel Thompson - Imperial College London
// with large use of code supplied by James Rosindell (Imperial College London)
// This file contains the datapoint class for usage in coalescence simulations.
// Version 2.01 removes any references to matrix vs grid x and y coordinates, instead␣
↪simplifying to a single coordinate system.
```

```
# define version1_01

/*************************************************************
                    DATA POINT OBJECT
 *************************************************************/
 #include <iostream>
using namespace std;
class Datapoint
{

private:
    unsigned long xpos;
    // x position
    unsigned long ypos;
    // y position
    long xwrap;
    // number of wraps of x around the torus
    long ywrap;
    // number of wraps of y around the torus
    unsigned long iNext;
    // the next individual in the loop of those that have the same xypos
//  unsigned long last; // removed as of version 3.1
//  // the last individual in the loop 0 means the only one
    unsigned long mpos;
    // points to the position in output of this lineage
    unsigned short listpos;
    // points to the position in the SpeciesList file.
    unsigned short nwrap;

    // the max-min number
    double dMinmax;
public:

    Datapoint() : xpos(0),ypos(0),xwrap(0),ywrap(0),iNext(0),mpos(0),listpos(0),
↪nwrap(0),dMinmax(0)
    {

    }

    ~Datapoint()
    {

    }

    void setup(unsigned long x , unsigned long y , long xwrapin, long ywrapin,␣
↪unsigned long matrix_position, unsigned short listposin, double dMinmaxin)
    {
        xpos = x;
        ypos = y;
        xwrap = xwrapin;
        ywrap = ywrapin;
        iNext = 0;
//      last =0;// removed as of version 3.1
        mpos = matrix_position;
        listpos = listposin;
        nwrap = 0;
        dMinmax = dMinmaxin;
    }
```

```
    void setup(Datapoint datin)
    {
        xpos = datin.getXpos();
        ypos = datin.getYpos();
        xwrap = datin.getXwrap();
        ywrap = datin.getYwrap();
        iNext = datin.getNext();
//      last = datin.get_last(); // removed as of version 3.1
        mpos = datin.getMpos();
        listpos = datin.getListpos();
        nwrap = datin.getNwrap();
        dMinmax = datin.getMinmax();
    }


    void setMpos(unsigned long z)
    {
        mpos = z;
    }
    void setNext(unsigned long x)
    {
        iNext = x;
    }

    void setListpos(unsigned short l)
    {
        listpos = l;
    }
    void setNwrap(unsigned short n)
    {
        nwrap = n;
    }

    void setMinmax(double d)
    {
        dMinmax = d;
    }

    void subtractListpos()
    {
        listpos --;
    }
    // note that position variables (8 of them) are set by lineage move routines below

    unsigned long getXpos()
    {
        return xpos;
    }

    unsigned long getYpos()
    {
        return ypos;
    }

    long getXwrap()
    {
        return xwrap;
```

```
    }

    long getYwrap()
    {
        return ywrap;
    }

    unsigned long getMpos()
    {
        return mpos;
    }

    unsigned long getNext()
    {
        return iNext;
    }

    // removed as of version 3.1
//  unsigned long get_last()
//  {
//      return last;
//  }
    unsigned short getListpos()
    {
        return listpos;
    }

    unsigned short getNwrap()
    {
        return nwrap;
    }

    double getMinmax()
    {
        return dMinmax;
    }

    void decreaseNwrap()
    {
        try
        {
            if(nwrap==0)
            {
                throw out_of_range("ERROR_DATA_001: Trying to decrease  nwrap less␣
↪than 0.");
            }
            else
            {
                nwrap --;
            }
        }
        catch(out_of_range& oor)
        {
            cerr << oor.what() << endl;
        }
    }
    // routines
    // removed the move and checkpos routines as they are no longer relevant.
```

```
    void setEndpoint(long x, long y, long xwrapin, long ywrapin)
    {
        xpos = x;
        ypos = y;
        xwrap = xwrapin;
        ywrap = ywrapin;
    }

    friend ostream& operator<<(ostream& os, const Datapoint& d)
    {
        os << d.xpos << "," << d.ypos  << "," << d.xwrap << "," << d.ywrap << "," <<␣
↪d.iNext << "," << d.mpos << "," << d.listpos << "," << d.nwrap<< ",";
        os << d.dMinmax << ",";
        return os;
    }

    friend istream& operator>>(istream& is, Datapoint& d)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        char delim;
        //cout << "datapoint" << endl;
        is >> d.xpos >> delim >> d.ypos  >> delim >> d.xwrap >> delim >> d.ywrap >>␣
↪delim >> d.iNext >> delim >> d.mpos >> delim >> d.listpos >> delim >> d.nwrap>>␣
↪delim;
        is >> d.dMinmax >> delim;
        return is;
    }
};
```

Includes

- iostream

Included By

- *File Tree.h*

Classes

- *Class Datapoint*

Defines

- *Define version1_01*

File Fattaildeviate.h

Definition (`Fattaildeviate.h`)

Program Listing for File Fattaildeviate.h

- Return to documentation for *File Fattaildeviate.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for full license details.

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 5277
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-8
#define RNMX (1.0-EPS)

# include <stdio.h>
# include <string>
# include <math.h>

# include <vector>
# include <iostream>
# include <fstream>

using namespace std;

class NRrand {

private:
    long idum;
    int j;
    long k;
    long idum2;
    long iy;
    long iv[NTAB];
    double temp;
    bool seeded;

    double lastresult;
    bool normflag;
    // for the L value of the dispersal kernel (the width - does not affect the shape).
    double dLvalue;
    // for the zfat value of the dispersal kernel (affects the shape).
    double zfat;
```

```
public:

    NRrand()
    {
        seeded = false;
        normflag = true;
    }

    void setSeed(long seed)
    {
        if (!seeded)
        {
        idum2 = 123456789;
        iy = 0;
        idum = seed;
        if (idum < 1) idum=1;
        //Be sure to prevent idum = 0.
        idum2=(idum);
        for (j=NTAB+7;j>=0;j--)
        {
            //Load the shuffle table (after 8 warm-ups).
            k=(idum)/IQ1;
            idum=IA1*(idum-k*IQ1)-k*IR1;
            if (idum < 0) idum += IM1;
            if (j < NTAB) iv[j] = idum;
          }
        iy=iv[0];
        seeded = true;
        }
    }

    double d01()
    {
        k=(idum)/IQ1;
        //Start here when not initializing.
        idum=IA1*(idum-k*IQ1)-k*IR1;
        //Compute idum=(IA1*idum) % IM1 without overflows by Schrage's method.
        if (idum < 0) idum += IM1;
        k=idum2/IQ2;
        idum2=IA2*(idum2-k*IQ2)-k*IR2;
        //Compute idum2=(IA2*idum) % IM2 likewise.
        if (idum2 < 0) idum2 += IM2;
        j=iy/NDIV;
        //Will be in the range 0..NTAB-1.
        iy=iv[j]-idum2;
        //Here idum is shuffled, idum and idum2 are combined to generate output.
        iv[j] = idum;
        if (iy < 1) iy += IMM1;
        if ((temp=AM*iy) > RNMX)
        {
            //cout << "random call = " << "RNMAX" << "\n";
            return RNMX; //Because users don't expect endpoint values.
        }
        else
        {

            return temp;
```

```
        }
    }

    long i0(long max)
    {
        //long temp = (long(d01()*(max+1)));
        //cout << "random call = " << temp << "\n";
        return (long(d01()*(max+1)));
    }

     double norm()
    {
        if (normflag)
        {
        double r2 = 2;
        double xx;
        double yy;
        while (r2>1)
        {
            xx=2.0*d01()-1.0;
            yy=2.0*d01()-1.0;
            r2=(xx*xx)+(yy*yy);
        }
        double fac=sqrt(-2.0*log(r2)/r2);
        lastresult = xx*fac;
        double result = yy*fac;
        normflag = false;
        return result;
          }
        else
        {
            normflag = true;
            return lastresult;
        }
    }


    void setDispersalParams(double zfatin, double dLvaluein)
    {
        zfat = zfatin;
        dLvalue = dLvaluein;
    }


    double fattail(double z)
    {
        double result;
        result = pow((pow(d01(),(1.0/(1.0-z)))-1.0),0.5);
        return result;
    }

    // this new version corrects the 1.0 to 2.0 and doesn't require the values to be
↪passed every time.
    double fattail()
    {
        double result;
        result = (dLvalue * pow((pow(d01(),(2.0/(2.0-zfat)))-1.0),0.5));
        return result;
```

```
    }

    double direction()
    {
        double xx = 1.0 , yy = 1.0;
        while (xx*xx+yy*yy>1.0)
        {
        xx = d01();
        yy = d01();
        }
        return pow((xx/yy),2.0);
    }

    bool event(double probin)
    {
        if (probin < 0.000001)
        {
            if (d01() <= 0.000001)
            {
                return (event(probin * 1000000.0));
            }
            else
            {
                return false;
            }
        }
        else
        {
            if (probin > 0.999999)
            {
                return (!(event(1.0-probin)));
            }
            else
            {
                return (d01() <= probin);
            }
        }
    }

    // to reconstruct distribution, use x = fattail/sqrt(1+direction) , y = fattail/
↪sqrt(1+(direction^-1))

    friend ostream& operator<<(ostream& os,const NRrand& r)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;

        os << r.idum << ",";
        os << r.j << ",";
        os << r.k << ",";
        os << r.idum2 << ",";
        os << r.iy << ",";
        for(int i=0;i<NTAB;i++)
        {
            os << r.iv[i] << ",";
        }
        os << r.temp << ",";
        os << r.seeded << ",";
        os << r.lastresult << ",";
```

```
        os << r.normflag << "," << r.dLvalue << "," << r.zfat << ",";
        return os;
    }

    friend istream& operator>>(istream& is, NRrand& r)
    {
        char delim;
        //double temp1,temp2;
        //is << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        is >> r.idum;
        is >> delim;
        is >> r.j;
        is >> delim;
        is >> r.k;
        is >> delim;
        is >> r.idum2;
        is >> delim;
        is >> r.iy;
        is >> delim;
        for(int i=0;i<NTAB;i++)
        {
            is >> r.iv[i];
            is >> delim;
        }
        is >> r.temp;
        is >> delim;
        is >> r.seeded;
        is >> delim;
        is >> r.lastresult;
        is >> delim;
        is >>r.normflag;
        is >> delim >> r.dLvalue >> delim >> r.zfat >> delim;
        return is;
    }
};
```

Includes

- fstream
- iostream
- math.h
- stdio.h
- string
- vector

Included By

- *File Tree.h*

Classes

- *Class NRrand*

Defines

- *Define AM*
- *Define EPS*
- *Define IA1*
- *Define IA2*
- *Define IM1*
- *Define IM2*
- *Define IMM1*
- *Define IQ1*
- *Define IQ2*
- *Define IR1*
- *Define IR2*
- *Define NDIV*
- *Define NTAB*
- *Define RNMX*

File main.cpp

Definition (`main.cpp`)

Program Listing for File main.cpp

- Return to documentation for *File main.cpp*

```
//Copyright: 2015, Samuel Thompson, thompsonsed@gmail.com
// License: BSD-3
//All rights reserved.
//
//Redistribution and use in source and binary forms, with or without modification,␣
↪are permitted provided that the following conditions are met:
//
//1. Redistributions of source code must retain the above copyright notice, this list␣
↪of conditions and the following disclaimer.
//
//2. Redistributions in binary form must reproduce the above copyright notice, this␣
↪list of conditions and the following disclaimer in the documentation and/or other␣
↪materials provided with the distribution.
//
//3. Neither the name of the copyright holder nor the names of its contributors may␣
↪be used to endorse or promote products derived from this software without specific␣
↪prior written permission.
```

```
//THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY␣
↪EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED␣
↪WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.␣
↪IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,␣
↪INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT␣
↪NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR␣
↪PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,␣
↪WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)␣
↪ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE␣
↪POSSIBILITY OF SUCH DAMAGE.

//
#ifdef DEBUG
#ifndef verbose
#define verbose
#endif
#endif
#include <stdio.h>
#include "Setup.h"
// #define pristine_mode // not required unless you experience problems.
// This performs a more thorough check after each move operation.
// Currently, it will also check that the pristine state value is greater than the␣
↪returned value within every map cell.
// Note that this may cause problems if the pristine state is not the state with the␣
↪highest number of individuals.



/*************************************************************
        MAIN ROUTINE AND COMMAND LINE ARG ROUTINES

 *************************************************************/



int main(int argc, char *argv[])
{
    /*
    // COMMAND LINE ARGUMENTS
    // The argument argc is the number of arguments supplied to the program␣
↪(automatically generated by c++)
    // The arguments 1 to N are as follows
    // 1: the task_iter used for setting the seed.
    // 2: the sample grid x dimension
    // 3: the sample grid y dimension
    // 4: the fine map file relative path.
    // 5: the fine map x dimension
    // 6: the fine map y dimension
    // 7: the fine map x offset
    // 8 the fine map y offset
    // 9: the coarse map file relative path.
    // 10: the coarse map x dimension
    // 11: the coarse map y dimension
    // 12: the coarse map x offset
    // 13: the coarse map y offset
    // 14: the scale of the coarse map compared to the fine (10 means resolution of␣
↪coarse map = 10 x resolution of fine map)
```

```
    // 15: the output directory
    // 16: the speciation rate.
    // 17: the dispersal distance (zfat).
    // 18: the deme size
    // 19: the deme sample size (as a proportion of deme size)
    // 20: the time to run the simulation (in seconds).
     // 21: lambda - the relative cost of moving through non-forest
    // 22: the_task - for referencing the specific task later on.
    // 23: the minimum number of species the system is known to contain.
    // 24: the pristine fine map file to use
    // 25: the pristine coarse map file to use
    // 26: the rate of forest change from pristine
    // 27: the time (in generations) since the pristine forest was seen.
    // 28: the dispersal L value (the width of the kernel.
    // 29: the sample mask, with binary 1:0 values for areas that we want to sample␣
↪from. If this is not provided then this will default to mapping the whole area.
    // 30:  the link to the file containing every generation that the list should be␣
↪expanded. This should be in the format of a list.
    // 31- onwards: speciation rates to apply after the simulation is complete.
      */
    #ifndef verbose
    openLogFile(false);
    #endif
    //freopen("Logs/Log_")
      bool fullmode = false;
     try
     {
         vector<string> comargs;
        try
        {
            // Import the command line arguments and parse the arguments correctly.
            importArgs(argc,argv,comargs);
            fullmode = parseArgs(argc,comargs);
//          cout << fullmode << endl;
        }
        catch(Main_Exception& me)
        {
            cerr << me.what() << endl;
            exit(-1);
        }
        // Add the command line variables to the Mapvars structure.
        try
        {
            if(fullmode)
            {
                checkFolders(comargs[4],comargs[9],comargs[24],comargs[25],
↪comargs[15],comargs[29]);
            }
            else
            {
//              cout << 'test' << endl;
                if(comargs[3] == "null")
                {
                    throw Fatal_Exception("ERROR_MAIN_009: FATAL. Map file cannot be␣
↪null.");
                }
                if(comargs[4] == "null")
                {
```

```
                    throw Fatal_Exception("ERROR_MAIN_009: FATAL. Output folder␣
↪cannot be null.");
                }
//              for(unsigned i = 0; i < comargs.size(); i ++ )
//              {
//                  cout << comargs[i] << endl;
//              }
                doesExist(comargs[3]);
                doesExist(comargs[4]);
                doesExistNull(comargs[12]);
            }
        }
        catch(Fatal_Exception& fe)
        {
            cerr << fe.what() << endl;
            exit(-1);
        }
        long the_seed = stoi(comargs[1]);

        // import the command line arguments (or config file parameters) in to the␣
↪Mapvars object.
        // This object is passed on to the tree object for integration in to the␣
↪whole simulation.
        // Create the tree object which contains the whole simulation
        Tree tree;
        tree.setSeed(the_seed); // set the seed - removed the +161 as this can now be␣
↪incorporated at a higher level, allowing for cleaner control.
        if(fullmode)
        {
            cout << "INITIALISING...." << endl << "seed: " << the_seed << endl <<
↪"Speciation Rate: " << comargs[16]<< endl << "Dispersal (z,L): " << "(" <
↪<comargs[17]<<"," << comargs[28] << ")" <<endl;
        }
        else
        {
            cout << "INITIALISING...." << endl << "seed: " << the_seed << endl <<
↪"Speciation Rate: " << comargs[5]<< endl << "Dispersal (z,L): " << "(" <<comargs[6]<
↪<"," << comargs[7] << ")" <<endl;
        }
        // Setup the simulations.
        tree.setup(comargs,fullmode);
        // Check for previous unfinished simulations with the same seed number.
        bool simCheck = false;
        if(fullmode)
        {
            simCheck = checkSims(stoi(comargs[22]),the_seed,comargs[15]);
        }
        else
        {
            simCheck = checkSims(stoi(comargs[2]),the_seed,comargs[4]);
        }
        tree.simResume(simCheck);  //will do nothing if check is false. Otherwise␣
↪will load the paused simulation into RAM.
        // Run the simulation - the single longest and most important step.
        bool isComplete = tree.runSim();
        if ( isComplete )
        {
//          cout << "size: " << argc << " alt: " << comargs.size() << endl;
```

```
            getSR(comargs.size(), comargs,tree);
        }
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        return -1;
    }
    catch(Map_Fatal_Exception& mfe)
    {
        cerr << mfe.what() << endl;
        return -1;
    }
    catch(const exception &e)
    {
        cerr << e.what() << endl;
    }
    #ifndef verbose
    fclose(stdout);
    #endif
    return 0;
}
```

Includes

- `Setup.h` (*File Setup.h*)
- `stdio.h`

Functions

- *Function main*

File Map.cpp

Definition (`Map.cpp`)

Program Listing for File Map.cpp

- Return to documentation for *File Map.cpp*

```
// This file is part of NECSim project which is released under BSD-3 license.
// See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause)␣
↪for full license details
#include "Map.h"
#include "Setup.h"

void Map::setDims(Mapvars mapvarsin)
{
    if(!checksetdim)  // checks to make sure it hasn't been run already.
    {
        mapvars = mapvarsin;
        deme = mapvarsin.deme;
```

```
        xdim = mapvarsin.vargridxsize;
        ydim = mapvarsin.vargridysize;
        scale = mapvarsin.varcoarsemapscale;
        nUpdate = 0;
        checksetdim = true;
        dUpdateTime = 0;
        dPristine = mapvarsin.dPristine;
        dForestTransform = mapvarsin.dForestTransform;
    }
    else
    {
        cerr << "ERROR_MAP_001: Dimensions have already been set" << endl;
        return;
    }
    return;
}

bool Map::checkMapExists()
{
    for(unsigned int i = 0; i < mapvars.configs.getSectionOptionsSize(); i++)
    {
        string tmppath = mapvars.configs[i].getOption("path");
        if(!doesExistNull(tmppath))
        {
            return (false);
        }
    }
    return (true);
}

void Map::calcFineMap()
{
    string fileinput = mapvars.finemapfile;
    long mapxsize = mapvars.varfinemapxsize;
    long mapysize = mapvars.varfinemapysize;
    if(!checksetdim)  // checks that the dimensions have been set.
    {
        throw Map_Exception("ERROR_MAP_002: dimensions not set.");
    }
    // Note that the default "null" type is to have 100% forest cover in every cell.
    Matrix<float> toret;
    toret.SetSize(mapysize, mapxsize);
    fine_map.SetSize(mapysize, mapxsize);
    if(fileinput == "null")
    {
        for(int i = 0; i < mapxsize; i++)
        {
            for(int j = 0; j < mapysize; j++)
            {
                toret[j][i] = 1.0;
            }
        }
    }
    else  // There is a map to read in.
    {
        toret.import(fileinput);
    }
    // cout << toret << endl;
```

```
    for(int i = 0; i < mapxsize; i++)
    {
        for(int j = 0; j < mapysize; j++)
        {
            fine_map[j][i] = round(toret[j][i] * deme);
//          cout << fine_map[j][i] << endl;
        }
    }
    // cout << "deme: " << deme << endl;
    return;
    //*/
}

void Map::calcPristineFineMap()
{
    string fileinput = mapvars.pristinefinemapfile;
    long mapxsize = mapvars.varfinemapxsize;
    long mapysize = mapvars.varfinemapysize;
    if(!checksetdim)  // checks that the dimensions have been set.
    {
        throw Map_Exception("ERROR_MAP_002: dimensions not set.");
    }
    // Note that the default "null" type is to have 100% forest cover in every cell.
    Matrix<float> toret;
    toret.SetSize(mapysize, mapxsize);
    pristine_fine_map.SetSize(mapysize, mapxsize);
    if(fileinput == "null")
    {
        for(int i = 0; i < mapxsize; i++)
        {
            for(int j = 0; j < mapysize; j++)
            {
                toret[j][i] = 1.0;
            }
        }
    }
    else  // There is a map to read in.
    {
        toret.import(fileinput);
    }
    // cout << toret << endl;
    for(int i = 0; i < mapxsize; i++)
    {
        for(int j = 0; j < mapysize; j++)
        {
            pristine_fine_map[j][i] = round(toret[j][i] * deme);
        }
    }
    // cout << "deme: " << deme << endl;
    return;
}

void Map::calcCoarseMap()
{
    string fileinput = mapvars.coarsemapfile;
    long mapxsize = mapvars.varcoarsemapxsize;
    long mapysize = mapvars.varcoarsemapysize;
    if(!checksetdim)  // checks that the dimensions have been set.
```

```
    {
        throw Map_Exception("ERROR_MAP_003: dimensions not set.");
    }
    // Note that the default "null" type for the coarse type is to have non-forest in
↪every cell.
    Matrix<float> toret;
    toret.SetSize(mapysize, mapxsize);
    coarse_map.SetSize(mapysize, mapxsize);
    if(fileinput == "null")
    {
        for(int i = 0; i < mapxsize; i++)
        {
            for(int j = 0; j < mapysize; j++)
            {
                toret[j][i] = 0.0;
            }
        }
    }
    else  // There is a map to read in.
    {
        toret.import(fileinput);
    }
    for(int i = 0; i < mapxsize; i++)
    {
        for(int j = 0; j < mapysize; j++)
        {
            coarse_map[j][i] = round(toret[j][i] * deme * scale * scale);
        }
    }
    return;
}

void Map::calcPristineCoarseMap()
{
    string fileinput = mapvars.pristinecoarsemapfile;
    long mapxsize = mapvars.varcoarsemapxsize;
    long mapysize = mapvars.varcoarsemapysize;
    if(!checksetdim)  // checks that the dimensions have been set.
    {
        throw Map_Exception("ERROR_MAP_003: dimensions not set.");
    }
    // Note that the default "null" type for the coarse type is to have non-forest in
↪every cell.
    Matrix<float> toret;
    toret.SetSize(mapysize, mapxsize);
    pristine_coarse_map.SetSize(mapysize, mapxsize);
    if(fileinput == "null")
    {
        for(int i = 0; i < mapxsize; i++)
        {
            for(int j = 0; j < mapysize; j++)
            {
                toret[j][i] = 1.0;
            }
        }
    }
    else  // There is a map to read in.
    {
```

```
        toret.import(fileinput);
    }
    for(int i = 0; i < mapxsize; i++)
    {
        for(int j = 0; j < mapysize; j++)
        {
            pristine_coarse_map[j][i] = round(toret[j][i] * deme * scale * scale);
        }
    }

    return;
}

void Map::setTimeVars(double dPristinein, double dForestTransformin)
{
    dUpdateTime = 0;
    dPristine = dPristinein;
    dForestTransform = dForestTransformin;
}

void Map::calcOffset()
{
    if(mapvars.autocorrel_file != "null")
    {
        mapvars.setPristine(0);
    }
//  cout << mapvars.autocorrel_file << endl;
    if(fine_map.GetCols() == 0 || fine_map.GetRows() == 0)
    {
        throw Map_Exception("ERROR_MAP_004: fine map not set.");
    }
    if(coarse_map.GetCols() == 0 || coarse_map.GetRows() == 0)
    {
        coarse_map.SetSize(fine_map.GetRows(), fine_map.GetCols());
//      throw Map_Exception("ERROR_MAP_004: coarse map not set.");
    }
    finexoffset = mapvars.varfinemapxoffset;
    fineyoffset = mapvars.varfinemapyoffset;
    coarsexoffset = mapvars.varcoarsemapxoffset;
    coarseyoffset = mapvars.varcoarsemapyoffset;
    scale = mapvars.varcoarsemapscale;
    // this is the location of the top left (or north west) corner of the respective␣
↪map
    // and the x and y distance from the top left of the grid object that contains␣
↪the initial lineages.
    finexmin = -finexoffset;
    fineymin = -fineyoffset;
    finexmax = finexmin + (fine_map.GetCols());
    fineymax = fineymin + (fine_map.GetRows());
    coarsexmin = -coarsexoffset - finexoffset;
    coarseymin = -coarseyoffset - fineyoffset;
    coarsexmax = coarsexmin + scale * (coarse_map.GetCols());
    coarseymax = coarseymin + scale * (coarse_map.GetRows());
    lambda = mapvars.lambda;
#ifdef DEBUG
    cout << "finex: " << finexmin << "," << finexmax << endl;
    cout << "finey: " << fineymin << "," << fineymax << endl;
    cout << "coarsex: " << coarsexmin << "," << coarsexmax << endl;
```

```
    cout << "coarsey: " << coarseymin << "," << coarseymax << endl;
    cout << "offsets: "
         << "(" << finexoffset << "," << fineyoffset << ")(" << coarsexoffset << "," <
↪< coarseyoffset << ")" << endl;
    cout << "pristine fine file: " << pristine_fine_map << endl;
    cout << "pristine coarse file: " << pristine_coarse_map << endl;
#endif
    //      cout << "fine variables: " << finexmin << "," << finexmax << endl;
    //      cout << "coarse variabes: " << coarsexmin << "," << coarsexmax << endl;
    if(finexmin < coarsexmin || finexmax > coarsexmax || (finexmax - finexmin) < xdim␣
↪|| (fineymax - fineymin) < ydim)
    {
        throw Map_Fatal_Exception(
            "ERROR_MAP_006: FATAL - fine map extremes outside coarse map or sample␣
↪grid larger than fine map");
    }
    return;
}

void Map::validateMaps()
{
    cout << "\rValidating maps..." << flush;
    if(fine_map.GetCols() == pristine_fine_map.GetCols() && fine_map.GetRows() ==␣
↪pristine_fine_map.GetRows() &&
       coarse_map.GetCols() == pristine_coarse_map.GetCols() && coarse_map.GetRows()␣
↪== pristine_coarse_map.GetRows())
    {
        cout << "\rValidating maps...map sizes okay" << flush;
    }
    else
    {
        throw Map_Fatal_Exception(
            "ERROR_MAP_009: Map validation failed - modern and pristine maps are not␣
↪the same dimensions.");
    }
    double dTotal = fine_map.GetCols() + coarse_map.GetCols();
    unsigned long iCounter = 0;
    for(int i = 0; i < fine_map.GetCols(); i++)
    {
        for(int j = 0; j < fine_map.GetRows(); j++)
        {
            if(fine_map[j][i] > pristine_fine_map[j][i])
            {
                cerr << "fine map: " << fine_map[j][i] << "pristine map: " <<␣
↪pristine_fine_map[j][i]
                     << endl;
                cerr << "x,y: " << i << "," << j << endl;
                throw Map_Fatal_Exception("ERROR_MAP_007: Map validation failed -␣
↪fine map value larger "
                                          "than pristine fine map value.");
            }
        }
        double dPercentComplete = 100 * ((double)(i + iCounter) / dTotal);
        if(i % 1000 == 0)
        {
            cout << "\rValidating maps..." << dPercentComplete << "%                "
↪<< flush;
        }
```

```
    }
    iCounter = fine_map.GetCols();
    for(int i = 0; i < coarse_map.GetCols(); i++)
    {
        for(int j = 0; j < coarse_map.GetRows(); j++)
        {
            if(coarse_map[j][i] > pristine_coarse_map[j][i])
            {
                cerr << "coarse map: " << coarse_map[j][i] << " pristine map: " <<␣
↪pristine_coarse_map[j][i]
                     << endl;
                cerr << "coarse map x+1: " << coarse_map[j][i + 1]
                     << " pristine map: " << pristine_coarse_map[j][i + 1] << endl;
                cerr << "x,y: " << i << "," << j << endl;
                throw Map_Fatal_Exception("ERROR_MAP_008: Map validation failed -␣
↪coarse map value larger "
                                          "than pristine coarse map value.");
            }
        }
        double dPercentComplete = 100 * ((double)(i + iCounter) / dTotal);
        if(i % 1000 == 0)
        {
            cout << "\rValidating maps..." << dPercentComplete << "%                 "
↪<< flush;
        }
    }
    cout << "\rValidating maps complete                                          " <<␣
↪endl;
}

void Map::updateMap(double generation)
{
    // only update the map if the pristine state has not been reached.
    if(!mapvars.bPristine)
    {
        if(mapvars.dPristine < generation)
        {
            fine_map = pristine_fine_map;
            nUpdate++;
            mapvars.setPristine(nUpdate);
            // pristine_fine_map = mapvars.pristinefinemapfile;
            // pristine_coarse_map = mapvars.pristinecoarsemapfile;
            dCurrent = dPristine;
            dPristine = mapvars.dPristine;
            dForestTransform = mapvars.dForestTransform;
            calcPristineCoarseMap();
            calcPristineFineMap();
        }
    }
}

#ifdef infinite_landscape
unsigned short Map::getVal(
    const double& x, const double& y, const long& xwrap, const long& ywrap, const␣
↪double& dCurrentGen)
{
    return deme;
}
```

```
#endif

#ifndef infinite_landscape
unsigned short Map::getVal(
    const double& x, const double& y, const long& xwrap, const long& ywrap, const
↪double& dCurrentGen)
{
    double currentTime = dCurrentGen - dCurrent;
    double xval, yval;
    xval = x + (xdim * xwrap);  //
    yval = y + (ydim * ywrap);
    //      // return 0 if the requested coordinate is completely outside the map
    if(xval < coarsexmin || xval >= coarsexmax || yval < coarseymin || yval >=
↪coarseymax)
    {
//                  cout << "OUTSIDE: " << endl;
        return 0;
    }
    if(xval < finexmin || xval >= finexmax || yval < fineymin ||
       yval >= fineymax)  // check if the coordinate comes from the coarse resolution
↪map.
    {
//                  cout <<"COARSE" << endl;
        //          double tmp;
        // take in to account the fine map offsetting
        xval += finexoffset;
        yval += fineyoffset;
        // take in to account the coarse map offsetting and the increased scale of
↪the larger map.
        xval = floor(double((xval + coarsexoffset) / scale));
        yval = floor(double((yval + coarseyoffset) / scale));
        double retval = 0;
        if(bPristine || pristine_coarse_map[yval][xval] == coarse_map[yval][xval])
        {
            retval = pristine_coarse_map[yval][xval];
        }
        else
        {
            retval = coarse_map[yval][xval] +
                     (dForestTransform *
                      ((pristine_coarse_map[yval][xval] - coarse_map[yval][xval]) /
↪dPristine) * currentTime);
        }
//          cout << retval << endl;
//          cout << retval << endl;
// Note that debug mode will throw an exception if the returned value is less than
↪the pristine state
#ifdef pristine_mode
        if(retval > pristine_coarse_map[yval][xval])
        {
            string ec =
                "Returned value greater than pristine value. Check file input. (or
↪disable this error before "
                "compilation.\n";
            ec += "pristine value: " + to_string((long long)pristine_coarse_
↪map[yval][xval]) +
                  " returned value: " + to_string((long long)retval);
            throw Map_Fatal_Exception(ec);
```

```
        }
#endif
        return floor(retval);
    }
    //      cout << "fine_map val: " << fine_map[yval][xval] << endl;
    // take in to account the fine map offsetting
    // this is done twice to avoid having all the comparisons involve additions.
    xval += finexoffset;
    yval += fineyoffset;
    double retval = 0;
    if(bPristine || pristine_fine_map[yval][xval] == fine_map[yval][xval])
    {
        retval = pristine_fine_map[yval][xval];
    }
    else
    {
//      cout << pristine_fine_map[yval][xval] << endl;
//      cout << fine_map[yval][xval] << endl;
        retval = fine_map[yval][xval] +
                 (dForestTransform * ((pristine_fine_map[yval][xval] - fine_
↪map[yval][xval]) / (dPristine)) *
                  currentTime);
    }
// cout <<fine_map[yval][xval] << "-"<< retval << endl;
// Note that debug mode will throw an exception if the returned value is less than␣
↪the pristine state
#ifdef pristine_mode
    if(retval > pristine_fine_map[yval][xval])
    {
        throw Map_Fatal_Exception("Returned value greater than pristine value. Check␣
↪file input. (or disable this "
                                  "error before compilation.");
    }
#endif
    return floor(retval);
}
#endif
unsigned long Map::getInitialCount(double dSample, Datamask& samplemask)
{
    unsigned long toret;
    toret = 0;
//  cout << xdim << ", " << ydim << endl;
    for(long i = 0; i < xdim; i++)
    {
        for(long j = 0; j < ydim; j++)
        {
            unsigned long add = 0;
            if(samplemask.getVal(i, j))
            {

                add = floor(dSample * getVal(i, j, 0, 0, 0));
//              cout << add << endl;
//              cout << getVal(i, j, 0, 0, 0) << endl;
            }
            toret += add;
        }
    }
    return toret;
```

```
}

bool Map::checkMap(const double& x, const double& y, const long& xwrap, const long&␣
↪ywrap, const double generation)
{
    //       cout << "CHECK: " << getVal(x,y,xwrap,ywrap) << endl;
    if(getVal(x, y, xwrap, ywrap, generation) == 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool Map::checkFine(const double& x, const double& y, const long& xwrap, const long&␣
↪ywrap)
{
    double tmpx, tmpy;
    tmpx = x + xwrap * xdim;
    tmpy = y + ywrap * ydim;
    if(tmpx < finexmin || tmpx >= finexmax || tmpy < fineymin || tmpy >= fineymax)
    {
        return false;
    }
    else
    {
        return true;
    }
}

void Map::convertCoordinates(double& x, double& y, long& xwrap, long& ywrap)
{
    //      cout << "CONVERT START: " << x << "," << y << "," << xwrap << "," <<␣
↪ywrap << endl;
    // quick check
    //      bool bCheck = false;
    //      if(x < -1 || y < -1)
    //      {
    //          bCheck = true;
    //          cout << "this is good" << endl;
    //      }
    xwrap += floor(x / xdim);
    ywrap += floor(y / ydim);
    x = x - xwrap * xdim;
    y = y - ywrap * ydim;
    //      if(bCheck)
    //      {
    //          cout << "x,y " << x << "," << y <<endl;
    //          cout << "x,y wrap " << xwrap << "," <<ywrap << endl;
    //
    //      }
    //      cout << "CONVERT END: " << x << "," << y << "," << xwrap << "," << ywrap <
↪< endl;
    return;
}
```

```
void Map::runDispersal(const double& dist,
                       const double& angle,
                       long& startx,
                       long& starty,
                       long& startxwrap,
                       long& startywrap,
                       bool& disp_comp,
                       const double& generation)
{
//      if(startxwrap < -1 || startywrap < -1)
//      {
//          cout << "good1!" << endl;
//          cout << startxwrap << endl;
//          cout << startywrap << endl;
//      }
// Checks that the start point is not out of matrix - this might have to be disabled␣
↪to ensure that when updating the
// map, it doesn't cause problems.
#ifdef pristine_mode
    if(!checkMap(startx, starty, startxwrap, startywrap, generation))
    {
        disp_comp = true;
        return;
    }
#endif

    // Different calculations for each quadrant to ensure that the dispersal reads␣
↪the probabilities correctly.
    double newx, newy;

    newx = startx + (xdim * startxwrap) + 0.5;
    newy = starty + (ydim * startywrap) + 0.5;
    if(lambda == 1)  // then nothing complicated is required and we can jump straight␣
↪to the final point.
    {
        newx += double(dist * cos(angle));
        newy += double(dist * sin(angle));
        // cout << "new x,y: " << newx << "," << newy << endl;
    }
    else  // we need to see which deforested patches we pass over
    {
        long boost;
        boost = 1;
        double cur_dist, tot_dist, l;
        cur_dist = 0;
        tot_dist = 0;
        // Four different calculations for the different quadrants.
        if(angle > 7 * M_PI_4 || angle <= M_PI_4)
        {
            // Continue while the dist travelled is less than the dist energy
            while(cur_dist < dist)
            {
                // Check if the starting position of the loop is in the fine map or␣
↪not.
                if(checkFine(newx, newy, 0, 0))
                {
                    // Keep the standard movement rate
                    boost = 1;
```

```
                }
                else
                {
                    // Accellerate the travel speed if the point is outside the fine␣
↪grid.
                    // Note this means that lineages travelling from outside the fine␣
↪grid to within the
                    // fine grid may
                    // see 1 grid's worth of approximation, rather than exact values.
                    // This is an acceptable approximation!
                    boost = deme;
                }

                // Add the value to the new x and y values.
                newx = newx + boost;
                newy = newy + boost * tan(angle);
                // Check if the new point is within forest.
                if(checkMap(newx, newy, 0, 0, generation))
                {
                    l = 1;
                }
                else
                {
                    l = lambda;
                }
                // Move forward different dists based on the difficulty of moving␣
↪through forest.
                cur_dist = cur_dist + l * boost * (1 / cos(angle));
                tot_dist = tot_dist + boost * (1 / cos(angle));
            }
        }
        else if(angle > 3 * M_PI_4 && angle <= 5 * M_PI_4)
        {
            while(cur_dist < dist)
            {
                if(checkFine(newx, newy, 0, 0))
                {
                    boost = 1;
                }
                else
                {
                    boost = deme;
                }
                // Add the change to the new x and y values.
                newx = newx - boost;
                newy = newy + boost * tan(M_PI - angle);
                if(checkMap(newx, newy, 0, 0, generation))
                {
                    l = 1;
                }
                else
                {
                    l = lambda;
                }
                cur_dist = cur_dist + boost * l * (1 / cos(M_PI - angle));
                tot_dist = tot_dist + boost * (1 / cos(M_PI - angle));
            }
        }
```

```
        else if(angle > M_PI_4 && angle <= 3 * M_PI_4)
        {
            while(cur_dist < dist)
            {
                if(checkFine(newx, newy, 0, 0))
                {
                    boost = 1;
                }
                else
                {
                    boost = deme;
                }
                // Add the change to the new x and y values.
                newx = newx + boost * tan(angle - M_PI_2);
                newy = newy + boost;
                if(checkMap(newx, newy, 0, 0, generation))
                {
                    l = 1;
                }
                else
                {
                    l = lambda;
                }
                cur_dist = cur_dist + l * boost / cos(angle - M_PI_2);
                tot_dist = tot_dist + boost / cos(angle - M_PI_2);
            }
        }
        else if(angle > 5 * M_PI_4 && angle <= 7 * M_PI_4)
        {
            //              cout << "...ang4..." <<  flush;
            while(cur_dist < dist)
            {
                if(checkFine(newx, newy, 0, 0))
                {
                    boost = 1;
                }
                else
                {
                    boost = deme;
                }
                newx = newx + boost * tan(3 * M_PI_2 - angle);
                newy = newy - boost;
                if(checkMap(newx, newy, 0, 0, generation))
                {
                    l = 1;
                }
                else
                {
                    l = lambda;
                }
                cur_dist = cur_dist + l * boost / cos(3 * M_PI_2 - angle);
                tot_dist = tot_dist + boost / cos(3 * M_PI_2 - angle);
            }
        }
        // Move the point back to get the exact placement
        if(checkMap(newx, newy, 0, 0, generation))
        {
            tot_dist = tot_dist - min(cur_dist - dist, (double(boost) - 0.001));
```

```
        }
        else
        {
            disp_comp = true;
        }
        newx = startx + 0.5 + tot_dist * cos(angle);
        newy = starty + 0.5 + tot_dist * sin(angle);
    }
    if(checkMap(newx, newy, 0, 0, generation))
    {
        long newxwrap, newywrap;
        newxwrap = 0;
        newywrap = 0;
        //          cout << "new x,y: " << newx << "," << newy << endl;
        //          cout <<"BEFORE CHECK: " <<endl;
        //          cout << "start " << newxwrap << endl;
        convertCoordinates(newx, newy, newxwrap, newywrap);
        //          cout << "end " << newxwrap << endl;

        //          if(newxwrap < -1 || newywrap < -1)
        //          {
        //              cout << "good2!" << endl;
        //              cout << newxwrap << endl;
        //              cout << newywrap << endl;
        //          }
        if(!checkMap(newx, newy, newxwrap, newywrap, generation))
        {
            throw Map_Fatal_Exception(string(
                "ERROR_MOVE_007: Dispersal attempted to non-forest. Check dispersal␣
↪function. Forest cover: " +
                to_string((long long)getVal(newx, newy, newxwrap, newywrap,␣
↪generation))));
        }
        //          cout << "new x,y: " << newx << "," << newy << endl;
        startx = newx;
        starty = newy;
        startxwrap = newxwrap;
        startywrap = newywrap;
        disp_comp = false;
    }
    else
    {
        // cout << "OUT!" << endl;
        // cout << "x,y : " << newx << "," << newy << endl;
        disp_comp = true;
    }
    //      if(startxwrap < -1 || startywrap < -1)
    //      {
    //          cout << "good2!" << endl;
    //          cout << startxwrap << endl;
    //          cout << startywrap << endl;
    //      }
    return;
};

void Map::clearMap()
{
    dCurrent = 0;
```

```
    checksetdim = false;
    bPristine = false;
}

void Map::printVars()
{
    cout << "fine x limits: " << finexmin << " , " << finexmax << endl;
    cout << "fine y limits: " << fineymin << " , " << fineymax << endl;
    cout << "fine map offset: " << finexoffset << " , " << fineyoffset << endl;
    cout << "coarse x limits: " << coarsexmin << " , " << coarsexmax << endl;
    cout << "coarse y limits: " << coarseymin << " , " << coarseymax << endl;
    cout << "x,y dims: " << xdim << " , " << ydim << endl;
}
```

Includes

- `Map.h` (*File Map.h*)
- `Setup.h` (*File Setup.h*)

File Map.h

Definition (`Map.h`)

Program Listing for File Map.h

- Return to documentation for *File Map.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details
#ifndef MAP
#define MAP
 # include <string>
 # include <stdio.h>
 #include <vector>
 # include <iostream>
 # include <fstream>
 # include <math.h>
 # include <stdexcept>
 # include "Config.h"
 # include <boost/filesystem.hpp>

 #include "Matrix.h"
 //#include <Setup.h>
 //# include "Fattaildeviate.cpp"
 using namespace std;



/*************************************************************
                    MAPVARS STRUCTURE
 *************************************************************/
```

```
struct Mapvars
{
    string finemapfile, coarsemapfile,outdirectory,pristinefinemapfile,
↪pristinecoarsemapfile, samplemaskfile;
     // for file naming purposes.
    long the_task;
    // the variables for the grid containing the initial individuals.
    unsigned long vargridxsize, vargridysize;
    // The fine map variables at the same resolution as the grid.
    unsigned long varfinemapxsize, varfinemapysize, varfinemapxoffset,␣
↪varfinemapyoffset;
    // the coarse map variables at a scaled resolution of the fine map.
    unsigned long varcoarsemapxsize, varcoarsemapysize, varcoarsemapxoffset,␣
↪varcoarsemapyoffset,varcoarsemapscale;
    unsigned long desired_specnum;
    // the relative cost of moving through non-forest
    double lambda;
    // the size of each square of habitat in numbers of individuals
    long deme;
    // the sample proportion,
     double deme_sample;
    // the speciation rate.
    long double  spec;
    // the fatness of the dispersal kernel.
    double zfat;
    // max time to run for
    double maxtime;
    // the number of generations since a pristine landscape was encountered.
    double dPristine;
    // the transform rate of the forest from pristine to modern forest.
    double dForestTransform;
    // the dispersal L value (the width of the fat-tailed kernel).
    double dLvalue;
    // file containing the points to record data from
    string autocorrel_file;
    // Stores the full list of configs imported from file
    ConfigOption configs;
    // Set to true if the completely pristine state has been reached.
    bool bPristine;


    void import(const vector<string> &comargs, bool fullmode)
    {
        finemapfile = "null";
        coarsemapfile = "null";
        pristinefinemapfile = "null";
        pristinecoarsemapfile = "null";
        if(fullmode)
        {
            vargridxsize = stol(comargs[2]);
            vargridysize = stol(comargs[3]);
            finemapfile = comargs[4];
            varfinemapxsize = stol(comargs[5]);
            varfinemapysize = stol(comargs[6]);
            varfinemapxoffset = stol(comargs[7]);
            varfinemapyoffset = stol(comargs[8]);
            coarsemapfile = comargs[9];
            varcoarsemapxsize = stol(comargs[10]);
```

```
            varcoarsemapysize = stol(comargs[11]);
            varcoarsemapxoffset = stol(comargs[12]);
            varcoarsemapyoffset = stol(comargs[13]);
            varcoarsemapscale = stol(comargs[14]);
            outdirectory = comargs[15];
            lambda = stod(comargs[21]);
            the_task = stol(comargs[22]);
            desired_specnum = stol(comargs[23]);
            pristinefinemapfile = comargs[24];
            pristinecoarsemapfile = comargs[25];
            dForestTransform = stod(comargs[26]);
            dPristine = stod(comargs[27]);
            deme = stol(comargs[18]);
            deme_sample = stod(comargs[19]);
            spec = stold(comargs[16]);
            zfat = stod(comargs[17]);
            maxtime = stod(comargs[20]);
            dLvalue = stod(comargs[28]);
            samplemaskfile = comargs[29];
            autocorrel_file = comargs[30];
        }
        else
        {
            // do the import of the values from combination of command-line arguments␣
↪and file.
            configs.setConfig(comargs[3],false,true);
            configs.parseConfig();
            vargridxsize = stol(configs.getSectionOptions("sample_grid", "x"));
            vargridysize = stol(configs.getSectionOptions("sample_grid", "y"));
            samplemaskfile = configs.getSectionOptions("sample_grid","mask");
            finemapfile = configs.getSectionOptions("fine_map", "path");
            varfinemapxsize = stol(configs.getSectionOptions("fine_map", "x"));
            varfinemapysize = stol(configs.getSectionOptions("fine_map", "y"));
            varfinemapxoffset = stol(configs.getSectionOptions("fine_map", "x_off"));
            varfinemapyoffset = stol(configs.getSectionOptions("fine_map", "y_off"));
            coarsemapfile = configs.getSectionOptions("coarse_map", "path");
            varcoarsemapxsize = stol(configs.getSectionOptions("coarse_map", "x"));
            varcoarsemapysize = stol(configs.getSectionOptions("coarse_map", "y"));
            varcoarsemapxoffset = stol(configs.getSectionOptions("coarse_map", "x_off
↪"));
            varcoarsemapyoffset = stol(configs.getSectionOptions("coarse_map", "y_off
↪"));
            varcoarsemapscale = stol(configs.getSectionOptions("coarse_map", "scale
↪"));
            pristinefinemapfile = configs.getSectionOptions("pristine_fine0", "path");
            pristinecoarsemapfile = configs.getSectionOptions("pristine_coarse0",
↪"path");
            outdirectory = comargs[4];
            the_task = stol(comargs[2]);
             //= stol(comargs[5]);
            zfat = stod(comargs[6]);
            dLvalue = stod(comargs[7]);
            deme = stol(comargs[8]);
            deme_sample = stod(comargs[9]);
            maxtime = stod(comargs[10]);
            lambda = stod(comargs[11]);
            autocorrel_file = comargs[12];
            spec = stod(comargs[5]);
```

```
//            cout << spec<< endl;
//            cout << comargs[5] << endl;
//            cout << "auto" << autocorrel_file << endl;
              desired_specnum = stol(comargs[13]);
//            printVars();
          }
      }

      void setPristine(unsigned int n)
      {
          bPristine = true;
          // Loop over each element in the config file (each line) and check if it is␣
↪pristine fine or pristine coarse.
          for(unsigned int i = 0; i < configs.getSectionOptionsSize(); i ++ )
          {

              if(configs[i].section.find("pristine_fine") == 0)
              {
                  // Then loop over each element to find the number, and check if it is␣
↪equal to our input number.
                  if(stol(configs[i].getOption("number")) == n)
                  {
                      bPristine = false;
                      pristinefinemapfile = configs[i].getOption("path");
                      dForestTransform = stod(configs[i].getOption("rate"));
                      dPristine = stod(configs[i].getOption("time"));
                  }
              }
              else if(configs[i].section.find("pristine_coarse") == 0)
              {
                  if(stol(configs[i].getOption("number")) == n)
                  {
                      bPristine = false;
                      pristinefinemapfile = configs[i].getOption("path");
                      // check matches
                      if(dForestTransform != stod(configs[i].getOption("rate")) ||␣
↪dPristine != stod(configs[i].getOption("time")))
                      {
                          cerr << "Forest transform values do not match between fine␣
↪and coarse maps. Using fine values." << endl;
                      }
                  }
              }
          }
      }
      void printVars()
      {
          cout << "task: " << the_task << endl;
          cout << "Fine input file: " << finemapfile  << endl;
          cout << "-dimensions: (" << varfinemapxsize << "," << varfinemapysize <<")"<<␣
↪endl;
          cout << "-offset: (" << varfinemapxoffset << "," << varfinemapyoffset << ")" <
↪< endl;
          cout << "Coarse input file: " << coarsemapfile  << endl;
          cout << "-dimensions: (" << varcoarsemapxsize << "," << varcoarsemapysize <<")
↪"<< endl;
          cout << "-offset: (" << varcoarsemapxoffset << "," << varcoarsemapyoffset <<
↪")" << endl;
```

```
        cout << "-scale: " << varcoarsemapscale << endl;
        cout << "Sample grid" << endl;
        cout << "-dimensions: (" << vargridxsize << "," << vargridysize << ")" <<␣
↪endl;
        cout << "-deme: " << deme << endl;
        cout << "-deme sample: " << deme_sample << endl;
        cout << "Output directory: " << outdirectory << endl;
        cout << "Lambda: " << lambda << endl;

    }

    friend ostream& operator<<(ostream& os,const Mapvars& m)
    {
        os << m.finemapfile << "\n" << m.coarsemapfile << "\n" << m.
↪pristinefinemapfile << "\n" << m.pristinecoarsemapfile << "\n" << m.samplemaskfile <
↪< "\n";
        os << m.the_task << "\n" <<  m.vargridxsize << "\n" << m.vargridysize << "\n"
↪<< m.varfinemapxsize << "\n" << m.varfinemapysize << "\n";
        os << m.varfinemapxoffset << "\n" << m.varfinemapyoffset << "\n" << m.
↪varcoarsemapxsize << "\n" << m.varcoarsemapysize << "\n" << m.varcoarsemapxoffset <
↪< "\n";
        os << m.varcoarsemapyoffset << "\n" << m.varcoarsemapscale << "\n" << m.
↪desired_specnum << "\n" << m.lambda << "\n" << m.deme << "\n" << m.deme_sample<< "\n
↪";
        os << m.spec << "\n" << m.zfat << "\n" << m.maxtime << "\n" << m.dPristine <<
↪"\n" << m. dForestTransform << "\n" << m.dLvalue << "\n" << m.autocorrel_file << "\n
↪";
        os << m.configs;
        return os;
    }

    friend istream& operator>>(istream& is, Mapvars& m)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        //char delim;
        //cout << "check" << endl;
        is >> m.finemapfile >> m.coarsemapfile >> m.pristinefinemapfile >> m.
↪pristinecoarsemapfile >> m.samplemaskfile ;
        is >> m.the_task >>  m.vargridxsize >> m.vargridysize >> m.varfinemapxsize >>␣
↪m.varfinemapysize ;
        is >> m.varfinemapxoffset >> m.varfinemapyoffset >> m.varcoarsemapxsize >> m.
↪varcoarsemapysize >> m.varcoarsemapxoffset ;
        is >> m.varcoarsemapyoffset >> m.varcoarsemapscale >> m.desired_specnum >> m.
↪lambda >> m.deme >> m.deme_sample;
        is >> m.spec >> m.zfat >> m.maxtime >> m.dPristine >> m. dForestTransform >>␣
↪m.dLvalue >> m.autocorrel_file;
        //cout << "2" << endl;
        is >> m.configs;
        //cout << "mate" << endl;
        return is;
    }
};

// Class which contains the Datamask object, telling us where to sample from within␣
↪the habitat map.
class Datamask
{
private:
```

```
    string inputfile; // the file to read in from
    bool bDefault;
public:
    Matrix<bool> sample_mask;
    Datamask()
    {
        bDefault = true;
    }

    void importDatamask(Mapvars& mapvarin)
    {
        inputfile = mapvarin.samplemaskfile;
//      cout << "inputfile: " << inputfile;
        if(inputfile == "null")
        {
            bDefault = true;
        }
        else
        {
            sample_mask.SetSize(mapvarin.vargridysize,mapvarin.vargridxsize);
            bDefault = false;
            sample_mask.import(inputfile);
        }
    }

    bool getVal(long x, long y)
    {
        if(bDefault)
        {
            return true;
        }
        else
        {
            return sample_mask[y][x];
        }
    }
};
/*************************************************************
                    MAP OBJECT
 *************************************************************/
 // Object containing both the maps (the coarse and fine version) and routines for␣
↪easy setting up and switching between the different coordinate systems.
 class Map
 {
private:
    // The map files which are read in (or generated if running with "null" as the␣
↪map file".
    // Pristine maps are meant for before any deforestation occured, whereas the␣
↪other maps are intended for modern day maps.
    // A linear transformation from modern to pristine maps is used, approaching the␣
↪dForestTransform variable times the difference between the pristine and modern maps.
    // Once the dPristine number of generations has been reached, the map will jump␣
↪to the pristine condition.
     Matrix<unsigned short> fine_map; // the finer grid for the area around the␣
↪sample area.
     Matrix<unsigned short> pristine_fine_map; // the pristine finer map.
     Matrix<unsigned short> coarse_map; // the coarser grid for the wider zone.
     Matrix<unsigned short> pristine_coarse_map; // the pristine coarser map.
```

```
    Mapvars mapvars; // for importing and storing the simulation set-up options.
    long finexmin,fineymin,coarsexmin,coarseymin; // the minimum values for each␣
↪dimension for offsetting.
    long finexmax,fineymax,coarsexmax,coarseymax; // the maximum values for each␣
↪dimension for offsetting.
    long finexoffset, fineyoffset, coarsexoffset, coarseyoffset; // the offsetting␣
↪of the map in FINE map units.
    int scale; // the scale of the coarse map compared with the smaller map.
    long xdim; // the length of the grid where the species start.
    long ydim; // the height of the grid where the species start.
    long deme;
    bool checksetdim; // for checking that the dimensions have been set before␣
↪attempting to import the maps.
    double lambda; // for setting the movement cost through forest.

   double dUpdateTime; // the last time the map was updated, in generations.
   double dForestTransform; // the rate at which the forest transforms from the␣
↪modern forest map to the pristine forest map. A value of 1 will give a smooth curve␣
↪from the present day to pristine forest.
   double dPristine; // the number of generations at which point the forest becomes␣
↪entirely pristine.
   double dCurrent; // the time the current map was updated.
   bool bPristine; // checks whether the simulation has already been set to the␣
↪pristine state.

    string NextMap;
    unsigned int nUpdate; // the number of updates to have occured.

public:
    Map()
    {
        checksetdim = false; // sets the check to false.
        bPristine = false;
        dCurrent = 0;
    }

    void setDims(Mapvars mapvarsin);

    bool checkMapExists();

    /*****************************************
     * CALC MAP FUNCTIONS
     *****************************************/

     void calcFineMap() ;

    void calcPristineFineMap() ;

    void calcCoarseMap();

    void calcPristineCoarseMap();

    // Map setters
    void setTimeVars(double dPristinein, double dForestTransformin);


    void calcOffset();
    /*****************************************
```

```
     * VALIDATE MAPS
     *****************************************/

    void validateMaps();

    /*****************************************
     * CHANGE MAP FUNCTIONS
     *****************************************/

    void updateMap(double generation);

    bool isPristine()
    {
        return bPristine;
    }

    void setPristine(const bool &bPristinein)
    {
        bPristine = bPristinein;
    }
    //
    /*****************************************
     * GET VAL FUNCTIONS
     *****************************************/


    // Function for getting the val at a particular coordinate from either the coarse␣
↪or fine map
    // altered to use the current generation as well to determine the value.
    unsigned short getVal(const double &x,const double &y,const long &xwrap,const␣
↪long &ywrap, const double &dCurrentGen);

    unsigned long getInitialCount(double dSample, Datamask& samplemask);

    /*****************************************
     * CHECK MAP FUNCTIONS
     *****************************************/
     bool checkMap(const double &x,const double &y, const long &xwrap,const long &
↪ywrap,const double generation);


     bool checkFine(const double &x,const double &y, const long &xwrap,const long &
↪ywrap);


    void convertCoordinates(double &x, double &y, long &xwrap, long &ywrap);

    /*****************************************
     * MAIN DISPERSAL FUNCTION
     *****************************************/
    void runDispersal(const double &dist, const double &angle, long &startx ,long  &
↪starty , long &startxwrap, long &startywrap, bool &disp_comp, const double &
↪generation);


    friend ostream& operator<<(ostream& os,const Map& r)
    {
        os << r.mapvars << "\n" << r.finexmin << "\n" << r.finexmax << "\n" << r.
↪coarsexmin << "\n" << r.coarsexmax << "\n" << r.fineymin << "\n" << r.fineymax <<
↪"\n" << r.coarseymin << "\n" << r.coarseymax << "\n";
```

```
        os << r.finexoffset << "\n" << r.fineyoffset << "\n" << r.coarsexoffset << "\n
↪" << r.coarseyoffset << "\n" << r.scale << "\n" << r.xdim << "\n" << r.ydim << "\n"
↪<< r.deme << "\n" << r.checksetdim << "\n" << r.lambda << "\n";
        os << r.dUpdateTime << "\n" << r.dForestTransform << "\n" << r.dPristine <<
↪"\n" << r.dCurrent << "\n" << r.bPristine << "\n" << r.NextMap << "\n" << r.nUpdate
↪<< "\n";
        return os;
    }

    friend istream& operator>>(istream& is, Map& r)
    {
        //double temp1,temp2;
        //is << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        is >> r.mapvars >> r.finexmin >> r.finexmax >> r.coarsexmin >> r.coarsexmax >>
↪ r.fineymin >> r.fineymax >> r.coarseymin >> r.coarseymax;
        is >> r.finexoffset >> r.fineyoffset >> r.coarsexoffset >> r.coarseyoffset >>␣
↪r.scale >> r.xdim >> r.ydim >> r.deme >> r.checksetdim >> r.lambda;
        is >> r.dUpdateTime >> r.dForestTransform >> r.dPristine >> r.dCurrent >> r.
↪bPristine >> r.NextMap >> r.nUpdate;
        r.calcFineMap();
        r.calcCoarseMap();
        r.calcPristineFineMap();
        r.calcPristineCoarseMap();
        return is;
    }

    void printVars()
    ;

    void clearMap()
    ;
};
#endif
```

Includes

- Config.h (*File Config.h*)
- Matrix.h (*File Matrix.h*)
- boost/filesystem.hpp
- fstream
- iostream
- math.h
- stdexcept
- stdio.h
- string
- vector

Included By

- *File Tree.h*
- *File Map.cpp*

Classes

- *Struct Mapvars*
- *Class Datamask*
- *Class Map*

File Matrix.h

Definition (`Matrix.h`)

Program Listing for File Matrix.h

- Return to documentation for *File Matrix.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
// Matrix.cpp version 1.12
// Author - James Rosindell Leeds University, modified to include additional␣
↪functionality by Samuel Thompson, Imperial College London
// but with large usage of reference http://www.devarticles.com/c/a/Cplusplus/
↪Operator-Overloading-in-C-plus/1
//
# define version1_11
// # define version1_1

#ifndef MATRIX
# define MATRIX
# define null 0
//# include "stdiostream.h"
# include <stdio.h>
#include <iostream>
#include <sstream>
#include <fstream>
#include <cstdlib>
#include <cstring>
#include <stdexcept>
//#define NDEBUG // uncomment this to disable all asserts.
#include <assert.h>
#ifdef use_csv
#include<cmath>
#include <stdexcept>
#include "fast-cpp-csv-parser/csv.h"
#endif
#ifdef with_gdal
#include <gdal_priv.h>
#include <cpl_conv.h> // for CPLMalloc()
```

```
#endif
#include <stdint.h>
using namespace std;

// Array of data sizes for importing tif files.
const int gdal_data_sizes[] = {0,8,16,16,32,32,32,64};

// Template class Row
template<class T>
class Row {
    // private variables
private:
    unsigned long numCols; // stores the number of columns in the row
    T* row; // an array to store the row

    // public variables
    public:

    Row(int cols = 0):row(0)
    {
        SetRowSize(cols);
    }

    ~Row()
    {
        delete[] row;
    }

    Row(const Row &r):row(0)
    {
        SetRowSize(r.numCols);
        for (unsigned int i=0;i<numCols;i++)
        {
            row[i]=r.row[i];
        }
    }

    // setter for row size

    void SetRowSize(unsigned long n)
    {
        if(row)
        {
            delete[] row;
        }
        if (n>0)
        {
            row=new T[n];
            // FIX THIS LATER!!!!!  memset(row,0,sizeof(T)*n/sizeof(char));
            // The call to memset() just zeroes out the array after figuring out how
↪many bytes the row uses and
            // dividing this by the size of character, because memset() works in
↪terms of chars.
        }
        else
        {
            row=0;
        }
```

```
        numCols=n;
    }

    void changeSize(unsigned long n)
    {
        try
        {
            T* tmprow = new T[n];
            for(unsigned int i =0;i < numCols; i++)
            {
                tmprow[i] = row[i];
            }
            delete[] row;
            row = tmprow;
    //      tmprow = NULL;
        }
        catch(bad_alloc & ba)
        {
            cerr << ba.what() << endl;
            cerr << "numCols: " << numCols << endl;
            cerr << "n: " << n << endl;
            exit(EXIT_FAILURE);
        }
        numCols = n;
    }

    unsigned long size()
    {
        return numCols;
    }

    T& operator[](unsigned long column)
    {
        // assert(column<numCols);
        // check we are within bounds

      # ifdef version1_1
        if (column >= numCols)
        {
            column = numCols -1;
        }
        if (column < 0)
        {
            column = 0;
        }
      # endif
      # ifdef version1_11
        if(column <0 || column >= numCols)
        {
            string err = "ERROR_MAIN_013b: Tried to call an indices that was out of␣
↪range of the row. Check row size definition. numCols: " + to_string((long long)␣
↪numCols) + " index: " + to_string((long long) column);
            throw out_of_range(err);
        }
//      if (column < 0)
//        {
//          column = -1 * column;
//          column = column % numCols;
```

```
//           column = numCols - column;
//         }
        column = column % numCols;
      # endif

        return row[column];
    }

    // Overloading = operator
    Row&  operator=(const Row& r)
    {
        SetRowSize(r.numCols);
        for (unsigned int i=0;i<numCols;i++)
        {
            row[i]=r.row[i];
        }

        //we return a reference, but this time it's a reference to itself.
        //First we set the size of the current row equal to that of the source row,␣
↪then we copy its values.
        //There is an important note here. Notice that I'm using [] on the primitive␣
↪T array itself
        //--NOT the overloaded []s of Row. Remember that Row's [] returns a reference,
↪ thus if we had written row[i]=r[i],
        //we would get a row that references the exact same data in memory, so that␣
↪when we changed one the other would change
        //--this isn't what we want at all, so we need to access the raw data in the␣
↪Row class.

        return *this;
    }

    friend ostream& operator<<(ostream& os,const Row& r)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        os << r.numCols << ",";
        for(unsigned int c=0;c<r.numCols;c++)
            {
                os << r.row[c] << ",";
            }
        return os;
    }

    friend istream& operator>>(istream& is, Row& r)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        char delim;
        int n;
        is >> n;
        r.SetRowSize(n);
        is >> delim;
        for (unsigned int c=0;c<r.numCols;c++)
            {
                is >> r.row[c];
                is >> delim;
            }
        return is;
    }
```

```
};


// Template class Matrix as an array of rows
template<class T>
class Matrix {

protected:

    // number of rows and columns
    unsigned long numCols, numRows;
    // a matrix is an array of rows
    Row<T>* matrix;

public:

    Matrix(int rows=0, int cols=0): matrix(null)
    {
        SetSize(rows,cols);
    }

    Matrix(const Matrix& m): matrix(null)
    {
        SetSize(m.numRows,m.numCols);
        for (int r=0;r<numRows;r++)
        matrix[r]=Row<T>(m.matrix[r]);//assign to primitive array, NOT overloaded []--
↪to get a copy
    }

    // HAD TO FIX NASTY MEMEORY LEAK HERE SINCE DOWLOADING!!
    ~Matrix()
    {
        delete[] matrix;

    }

    void SetSize(int rows, int cols)
    {
        if (matrix)
        {
            delete[]matrix;
            //if (rows) delete[]matrix; - was in original code - seems silly so␣
↪changed to the above
        }
        if (cols > 0 && rows >0)
        {
            matrix=new Row<T>[rows];
            for (int i=0;i<rows;i++)
            {
                matrix[i].SetRowSize(cols);
            }
        }
        else
        {
        rows=null;
        }
        numCols=cols;
        numRows=rows;
```

```
    }

    int GetCols() const
    {
        return numCols;
    }

    int GetRows()  const
    {
        return numRows;
    }

    // This whole piece so far follows similarly to that of the Row Class
    // now we go on to overload all the operators

    Row<T>& operator[](unsigned long index)
    {

        //assert(index<numRows);
         # ifdef version1_1
        if (index >= numRows)
        {
            index = numRows -1;
        }
        if (index < 0)
        {
            index = 0;
        }
        # endif
    # ifdef version1_11
//      if (index < 0)
//      {
//              index = -1 * index;
//              index = index % numRows;
//              index = numRows - index;
//      }
//      These asserts were lazy - changed to exceptions and proper error reporting␣
↪now.
//      assert(index<numRows && "out of range - index greater than the number of rows
↪");
//      assert(index>=0 && "out of range - index less than 0");
        if(index <0 || index >= numRows)
        {
            string err = "ERROR_MAIN_013: Tried to call an indices that was out of␣
↪range of the matrix. Check matrix size definition. numRows: " + to_string((long␣
↪long) numRows) + " index: " + to_string((long long) index);
            throw out_of_range(err);
        }
        index = index % numRows;
        # endif

        return matrix[index];
    }
    Matrix& operator=(const Matrix& m)
    {
        SetSize(m.numRows,m.numCols);
        for (unsigned long r=0;r<numRows;r++)
        {
```

```
            matrix[r]=Row<T>(m.matrix[r]);
            //assign to primitive array, NOT overloaded []--to get a copy
        }
        return *this;
    }


    const Matrix operator+( const Matrix& m)
    {
        //Since addition creates a new matrix, we don't want to return a reference,␣
↪but an actual matrix object.

        //assert(numCols==m.numCols && numRows==m.numRows);
        int newnumcols , newnumrows ;
        if (numCols > m.numCols)
        {
            newnumcols = m.numCols;
        }
        else
        {
            newnumcols = numCols;
        }
        if (numRows > m.numRows)
        {
            newnumrows = m.numRows;
        }
        else
        {
            newnumrows = numRows;
        }

        Matrix result(newnumrows,newnumcols);
        for (int r=0;r<newnumrows;r++)
        {
            for (int c=0;c<newnumcols;c++)
            {
                result[r][c]=matrix[r][c]+m.matrix[r][c];
            }
        }
        return result;
    }

    const Matrix operator-( const Matrix& m)
    {

        //assert(numCols==m.numCols && numRows==m.numRows);
        int newnumcols , newnumrows ;
        if (numCols > m.numCols)
        {
            newnumcols = m.numCols;
        }
        else
        {
            newnumcols = numCols;
        }
        if (numRows > m.numRows)
        {
            newnumrows = m.numRows;
```

```
        }
        else
        {
            newnumrows = numRows;
        }

        Matrix result(newnumrows,newnumcols);
        for (int r=0;r<newnumrows;r++)
        {
            for (int c=0;c<newnumcols;c++)
            {
                result[r][c]=matrix[r][c]-m.matrix[r][c];
            }
        }
        return result;
    }

    Matrix& operator+=(const Matrix& m)
    {
        // don't want to create a new object
        // just add whatever is currently in it to the other matrix, and return a␣
↪reference to itself:

        //assert(numCols==m.numCols && numRows==m.numRows);
        int newnumcols , newnumrows ;
        if (numCols > m.numCols)
        {
            newnumcols = m.numCols;
        }
        else
        {
            newnumcols = numCols;
        }
        if (numRows > m.numRows)
        {
            newnumrows = m.numRows;
        }
        else
        {
            newnumrows = numRows;
        }

        for (int r=0;r<newnumrows;r++)
        {
            for (int c=0;c<newnumcols;c++)
            {
                matrix[r][c]+=m.matrix[r][c];
            }
        }
        return *this;
    }


    Matrix& operator-=( const Matrix& m) {
        // don't want to create a new object
        // just subtractthe other matrix from whatever is currently in it, and return␣
↪a reference to itself:
```

```
        //assert(numCols==m.numCols && numRows==m.numRows);
        int newnumcols , newnumrows ;
        if (numCols > m.numCols)
        {
            newnumcols = m.numCols;
        }
        else
        {
            newnumcols = numCols;
        }
        if (numRows > m.numRows)
        {
            newnumrows = m.numRows;
        }
        else
        {
            newnumrows = numRows;
        }

        for (int r=0;r<newnumrows;r++)
        {
            for (int c=0;c<newnumcols;c++)
            {
                matrix[r][c]-=m.matrix[r][c];
            }
        }
        return *this;
    }

    const Matrix operator*(const double s)
    {
        Matrix result(numRows,numCols);
        for (int r=0;r<numRows;r++)
        {
            for (int c=0;c<numCols;c++)
            {
                result[r][c]=matrix[r][c]*s;
            }
        }
        return result;
    }

        const Matrix operator*(Matrix& m)
    {
        //assert(numCols==m.numRows);
        int newnumcols;
        if (numCols > m.numRows)
        {
            newnumcols = m.numRows;
        }
        else
        {
            newnumcols = numCols;
        }

        Matrix result(numRows,m.numCols);
        for (int r=0;r<numRows;r++)
        {
```

```
            for (int c=0;c<m.numCols;c++)
            {
                for (int i=0;i<newnumcols;i++)
                {
                    result[r][c]+=matrix[r][i]*m[i][c];
                }
            }
        }
        return result;
    }


    // The streaming operators << and >> allow your object to be saved and restored␣
↪from any
    // stream, be it console, network, or file.
    // There is a slight additional challenge with these operators because we must␣
↪allow the stream access to our object's private data.
    // Therefore, these functions must be declared as friends inside the Matrix class.

    friend ostream& operator<<(ostream& os,const Matrix& m)
    {
      //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        for (unsigned long r=0;r<m.numRows;r++)
        {
            for (unsigned long c=0;c<m.numCols;c++)
            {
                os << m.matrix[r][c] << ",";
            }
            os << "\n";
        }
        return os;
    }

    friend istream& operator>>(istream& is, Matrix& m)
    {
        //is << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        char delim;
        for (unsigned long r=0;r<m.numRows;r++)
        {
//          string line;
//          getline(is, line);
//          if ( !is.good() )
//          {
//              break;
//          }
//          stringstream iss(line);
//
//          for (int c = 0; c < m.numRows; c++)
//          {
//              std::string val;
//              std::getline(iss, val, ',');
//              if ( !iss.good() )
//                  break;
//
//              std::stringstream convertor(val);
//              convertor >> m.matrix[r][c];
//          }
            for (unsigned long c=0;c<m.numCols;c++)
```

```
            {
                is >> m.matrix[r][c];
                is >> delim;
            }
        }
        return is;
    }

    bool symmetric()
    {
        bool result = true;
        for (int c=0;c<numCols;c++)
        {
            for (int r=0;r<c+1;r++)
            {
                if (matrix[r][c] != matrix[c][r])
                {
                    result = false;
                    break;
                }
            }
        }
        return result;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value,const␣
↪double & t)
    {
        matrix[y][x] = atof(value);
        return;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value, const␣
↪char & t)
    {
        matrix[y][x] = *value;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value, const␣
↪int8_t & t)
    {
        matrix[y][x] = *value;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value, const␣
↪uint8_t & t)
    {
        matrix[y][x] = *value;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value,const␣
↪bool & t)
    {
        matrix[y][x] = bool(atoi(value));
        return;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char* value,const␣
↪unsigned long &t)
```

```
    {
        matrix[y][x] = long(value);
        return;
    }

    void setValue(const unsigned long &x, const unsigned long &y, char * value, const␣
↪long &t)
    {
        matrix[y][x] = long(value);
        return;
    }

     void import(string filename)
     {
         if(filename.find(".csv") != string::npos)
         {
             importCsv(filename);
         }
         else if(filename.find(".tif") != string::npos)
         {
             #ifdef with_gdal
             importTif(filename);
             return;
             #endif
             cerr << "Tif file specified, but program not compiled with gdal␣
↪libraries. Please recompile  with  -D with_gdal or edit your code to include
↪#define with_gdal." << endl;
             exit(EXIT_FAILURE);
         }
         else
         {
            string s = "Type detection failed for " + filename + ". Check filename is␣
↪correct.";
            throw runtime_error(s);
         }
     }

    #ifdef with_gdal
    void importTif(string filename)
    {
        cout  << "\rImporting " << filename << "..." << "0%" << "                    " <
↪< flush;
         GDALDataset  *poDataset;
         GDALAllRegister();
         poDataset = (GDALDataset *) GDALOpen( filename.c_str(), GA_ReadOnly );
         if( poDataset == NULL )
        {
            string s = "File " + filename + " not found.";
            throw runtime_error(s);
        }
        GDALRasterBand  *poBand;
        int nBlockXSize, nBlockYSize;
        // Import the raster band 1
        poBand = poDataset->GetRasterBand( 1 );
        nBlockXSize = poDataset->GetRasterXSize();
        nBlockYSize = poDataset->GetRasterYSize();
        // Check sizes
                    //cerr << "X dimension: " << nBlockXSize << endl;
```

```
            //cerr << "Y dimension: " << nBlockYSize << endl;
        if ( (numCols != (unsigned long) nBlockXSize || numRows != (unsigned long)␣
↪nBlockYSize) || numCols == 0 || numRows == 0)
        {
            cerr << "Raster data size does not match inputted dimensions. Using␣
↪raster sizes." << endl;
            cerr << "Old dimensions: " << numCols << ", " << numRows << endl;
            cerr << "New dimensions: " << nBlockXSize << ", " << nBlockYSize << endl;
            SetSize(nBlockYSize,nBlockXSize);
        }
        // Check sizes match
        GDALDataType dt = poBand->GetRasterDataType();
        #ifdef DEBUG
        if(sizeof(T)*8 != gdal_data_sizes[dt])
        {
            cerr << "Object data size: " << sizeof(T)*8 << endl;
            cerr << "Tif data size: " << gdal_data_sizes[dt] << endl;
            throw runtime_error("Tif data type does not match object data size."); //␣
↪Should this be a warning? I think this should be a warning...
        }
        #endif
        // Iterate over rows, reading each data segment into the matrix row.
        for(unsigned int j = 0; j < numRows; j ++)
        {
            double dComplete = ((double)j/(double)numRows)*100;
            cout  << "\rImporting " << filename << "..." << dComplete << "%" << "
↪              " << flush;
            CPLErr r = poBand->RasterIO( GF_Read, 0, j, nBlockXSize, 1, &matrix[j][0],
↪ nBlockXSize, 1, dt,0, 0 );
            if(r == CE_Failure)
            {
                throw runtime_error("CPL error during tif import: CE_Failure.");
            }
        }
        cout  << "\rImporting " << filename << "...done!" << "                    " <<␣
↪endl;
    }
    #endif

    #ifdef use_csv
    void importCsv(string filename)
    {
        cout  << "\rImporting " << filename << "..." << "0%" << "                    " <
↪< flush;
        // LineReader option
        io::LineReader in(filename);
        // Keep track of whether we've printed to terminal or not.
        bool bPrint = false;
        // Initialies empty variable so that the setValue operator overloading works␣
↪properly.
        T type_reference = null;
        for(unsigned long i =0; i<numRows;i++)
        {
            char* line = in.next_line();
            if(line == nullptr)
            {
                if(!bPrint)
                {
```

```
                    cerr << "Input dimensions incorrect - read past end of file." <<␣
↪endl;
                    bPrint = true;
                }
                break;
            }
            else
            {
                char *dToken;
                dToken = strtok(line,",");
                for(unsigned long j = 0;j<numCols;j++)
                {
                    if(dToken == nullptr)
                    {
                        if(!bPrint)
                        {
                            cerr << "Input dimensions incorrect - read past end of␣
↪file." << endl;
                            bPrint = true;
                        }
                        break;
                    }
                    else
                    {
                        // This function is overloaded to correctly determine the␣
↪type of the template
                        setValue(j,i,dToken,type_reference);
                        dToken = strtok(NULL,",");
                    }
                }
                // output the percentage complete
                double dComplete = ((double)i/(double)numRows)*100;
                if(i%40==0)
                {
                    cout  << "\rImporting" << filename << "..." << floor(dComplete) <
↪< "%" << "                    " << flush;
                }
            }
        }
        cout  << "\rImporting" << filename << "..."  << "done!" << "            " <<␣
↪endl;
    }
    #endif
    #ifndef use_csv
    // OLD VERSION - currently unused)
    void importCsv(string filename)
    {
        cout  << "\rImporting" << filename << "..." << "0%" << "                  " <
↪< flush;
         ifstream inputstream;
         inputstream.open(filename.c_str());
        for(int j =0;j<numRows;j++)
        {
            string line;
            getline(inputstream,line);
//          cout << "got line" << endl;
//          cout << line << endl;
            istringstream iss(line);
```

```
//            cout << numCols << endl;
//            string sam;
//            cin >> sam;

//            getline(inputstream,line);
//            cout << line << endl;
//            cin >> sam;
              for(int i = 0; i<numCols;i++)
              {
                  char delim;
                  T val;
                  iss >> val >> delim;
                  matrix[j][i] =val;
//                cout << i  << "-"<< matrix[j][i] << endl;
              }
              double dComplete = ((double)j/(double)numRows)*100;
              cout  << "\rImporting" << filename << "..." << dComplete << "%" << "
↪               " << flush;
//            cout << j << endl;
//            iss.clear();
          }
          cout  << "\rImporting" << filename << "..."  << "done!" << "
↪           " << endl;
          inputstream.close();
          return;
      }
      #endif
};

#endif

// THE END OF THE CODE
```

Includes

- assert.h
- cstdlib
- cstring
- fstream
- iostream
- sstream
- stdexcept
- stdint.h
- stdio.h

Included By

- *File Tree.h*
- *File Map.h*

- *File Treelist.h*

Classes

- *Class Matrix*
- *Class Row*

Defines

- *Define MATRIX*
- *Define null*
- *Define version1_11*

Variables

- *Variable gdal_data_sizes*

File Setup.cpp

Definition (`Setup.cpp`)

Program Listing for File Setup.cpp

- Return to documentation for *File Setup.cpp*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
//
#include "Setup.h"

// Global variables
// store the log file name for access anywhere.
string log_name = "null";
// the old stdout
int saved_stdout;
#ifndef verbose

void openLogFile(bool append)
{
    // If verbose mode is not defined, we output to a log file instead of to the␣
↪terminal
    // Get the current time
    saved_stdout = dup(fileno(stdout));
    //dup2(stdout, 1);
    auto t = time(nullptr);
    auto tm = *localtime(&t);
    // get the time string
    ostringstream oss;
    oss << std::put_time(&tm, "%d-%m-%Y-%H:%M:%S");
```

```
    if(log_name == "null")
    {
        log_name = "Logs/Log_"+oss.str() + ".txt";
    }
    // Check that the Log folder exists, and create if necessary.
    if(!boost::filesystem::exists("Logs"))
    {
        if(!boost::filesystem::create_directory("Logs"))
        {
            cerr << "Cannot create log directory (check write access) - defaulting to␣
↪terminal." << endl;
        }
    }
    if(boost::filesystem::exists("Logs"))
    {
        // Open the log file for writing to.
        FILE * tmpfileptr;
        if(append)
        {
            tmpfileptr = freopen(log_name.c_str(), "a", stdout);
        }
        else
        {
            tmpfileptr = freopen(log_name.c_str(), "w", stdout);
        }
        if(stdout == nullptr || tmpfileptr == nullptr)
        {
            cerr << "Cannot create log file (check write access) - defaulting to␣
↪terminal." << endl;
            dup2(saved_stdout, fileno(stdout));
            close(saved_stdout);
        }
    }
    //cerr<< "logfile2 : " << log_name << endl;
}
#endif

long charconvertor(char charin)
{
    switch (charin)
    {
        case '0': return (0);
        case '1': return (1);
        case '2': return (2);
        case '3': return (3);
        case '4': return (4);
        case '5': return (5);
        case '6': return (6);
        case '7': return (7);
        case '8': return (8);
        case '9': return (9);
        default: return (-1);
    }
}

long jobConvertor(char* argin)
{
    long maxind = 0;
```

```
    while (charconvertor(argin[maxind]) != -1)
    {
        maxind ++;
    }
    long jobtoret = 0;
    long pow10 = 1;
    for (long i = maxind-1 ; i >=0 ; i --)
    {
        jobtoret += (pow10*charconvertor(argin[i]));
        pow10 = pow10*10;
    }
    return jobtoret;
}

//
bool checkSims(int the_task,int the_seed,string outdirect)
{
    cout << "Checking for unfinished simulations..." << flush;
    ifstream out;
    string file_to_open;
//  char file_to_open[100];
//  sprintf (file_to_open, "%s/Pause/Data_%i.csv",outdirect,int(the_task));
    file_to_open = outdirect + string("/Pause/Dump_active_") + to_string((unsigned␣
↪long long)the_task)+"_"+to_string((unsigned long long)the_seed) + string(".csv");
    out.open(file_to_open);
    if(out.good())
    {
        cout << "done!" << endl << "File found containing unfinished simulations." <<␣
↪endl;
        return(true);
    }
    else
    {
        cout << "done!" << endl << "No files found containing unfinished simulations.
↪" << endl;
        return(false);
    }
}


void checkFolders(string sFineMap, string sCoarseMap, string sFineMapPristine, string␣
↪sCoarseMapPristine, string sOutputFolder,string sSampleMask)
{
    cout << "Checking folder existance..." << flush;
    bool bFineMap, bCoarseMap, bFineMapPristine, bCoarseMapPristine, bOutputfolder,␣
↪bSampleMask;
    try
    {
        bFineMap = doesExistNull(sFineMap);
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        bFineMap = false;
    }
    try
    {
        bCoarseMap = doesExistNull(sCoarseMap);
```

```
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        bCoarseMap = false;
    }
    try
    {
        bFineMapPristine = doesExistNull(sFineMapPristine);
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        bFineMapPristine = false;
    }
    try
    {
        bCoarseMapPristine = doesExistNull(sCoarseMapPristine);
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        bCoarseMapPristine = false;
    }
    if(sOutputFolder != "null")
    {
        try
        {
            bOutputfolder = doesExist(sOutputFolder);
        }
        catch(runtime_error &re)
        {
            cerr << "Output folder does not exist... creating..." << flush;
            bOutputfolder = boost::filesystem::create_directory(sOutputFolder);
            if( bOutputfolder)
            {
                cerr << "done!" << endl;
            }
            else
            {
                cerr << endl << re.what() << endl;
            }
        }
    }
    else
    {
        throw Fatal_Exception("ERROR_MAIN_009: FATAL. Output folder cannot be null.");
    }
    try
    {
        bSampleMask = doesExistNull(sSampleMask);
    }
    catch(Fatal_Exception& fe)
    {
        cerr << fe.what() << endl;
        bSampleMask = false;
    }
    if(bFineMap && bCoarseMap && bFineMapPristine && bCoarseMapPristine &&␣
↪bOutputfolder && bSampleMask)
```

```
    {
        cout << "\rChecking folder existance...done!
↪                                " << endl;
        return;
    }
    else
    {
        throw Fatal_Exception();
    }

}

void runAsDefault(vector<string> &comargs)
{
    cout << "Setting default variables on small grid..." << flush;
    comargs.push_back("-f");
    comargs.push_back("1");
    comargs.push_back("10");
    comargs.push_back("10");
    comargs.push_back("null");
    comargs.push_back("150");
    comargs.push_back("150");
    comargs.push_back("25");
    comargs.push_back("25");
    comargs.push_back("null");
    comargs.push_back("2000");
    comargs.push_back("2000");
    comargs.push_back("500");
    comargs.push_back("500");
    comargs.push_back("100");
    comargs.push_back("Default/");
    comargs.push_back("0.000009");
    comargs.push_back("4");
    comargs.push_back("1");
    comargs.push_back("1");
    comargs.push_back("3600");
    comargs.push_back("1");
    comargs.push_back("0");
    comargs.push_back("100");
    comargs.push_back("null");
    comargs.push_back("null");
    comargs.push_back("0.5");
    comargs.push_back("20.0");
    comargs.push_back("1.0");
    comargs.push_back("null");
    comargs.push_back("null");
    comargs.push_back("0.000009");
    cout << "done!" << endl;
}
void runLarge(vector<string> &comargs)
{
    cout << "Setting default variables on large grid..." << flush;
    comargs.push_back("-f");
    comargs.push_back("1");
    comargs.push_back("500");
    comargs.push_back("500");
    comargs.push_back("null");
    comargs.push_back("500");
```

```
    comargs.push_back("500");
    comargs.push_back("0");
    comargs.push_back("0");
    comargs.push_back("null");
    comargs.push_back("100");
    comargs.push_back("100");
    comargs.push_back("2500");
    comargs.push_back("2500");
    comargs.push_back("100");
    comargs.push_back("Default/");
    comargs.push_back("0.00001");
    comargs.push_back("8");
    comargs.push_back("10");
    comargs.push_back("1");
    comargs.push_back("3600");
    comargs.push_back("1");
    comargs.push_back("1");
    comargs.push_back("50000");
    comargs.push_back("null");
    comargs.push_back("null");
    comargs.push_back("0.5");
    comargs.push_back("20.0");
    comargs.push_back("1.0");
    comargs.push_back("null");
    comargs.push_back("null");
    comargs.push_back("0.000009");
    cout << "done!" << endl;
}

void runXL(vector<string> &comargs)
{
    cout << "Setting default variables on large grid..." << flush;
    comargs.push_back("-f");
    comargs.push_back("1");
    comargs.push_back("6000");
    comargs.push_back("6400");
    comargs.push_back("null");
    comargs.push_back("34000");
    comargs.push_back("28000");
    comargs.push_back("8800");
    comargs.push_back("14800");
    comargs.push_back("null");
    comargs.push_back("24000");
    comargs.push_back("20000");
    comargs.push_back("10320");
    comargs.push_back("8080");
    comargs.push_back("10");
    comargs.push_back("Default/");
    comargs.push_back("0.0000001");
    comargs.push_back("2");
    comargs.push_back("49");
    comargs.push_back("0.2");
    comargs.push_back("21600");
    comargs.push_back("1");
    comargs.push_back("3");
    comargs.push_back("600");
    comargs.push_back("null");
    comargs.push_back("null");
```

```
    comargs.push_back("0");
    comargs.push_back("2.2");
    comargs.push_back("1.0");
    comargs.push_back("null");
    comargs.push_back("null");
    comargs.push_back("0.000009");
    cout << "done!" << endl;
}


//
//
void removeComOption(int &argc, vector<string> &comargs)
{
    // stupidly long list of possible arguments, but can't think of a better way to␣
↪check this.
    if(comargs[1] == "-d" || comargs[1] == "-D" ||  comargs[1] == "-dl" || ␣
↪comargs[1] == "-dL" ||  comargs[1] == " -Dl" ||  comargs[1] == "-DL" ||
        comargs[1] == "-dx" || comargs[1] == "-dX" ||  comargs[1] == "-DX" || ␣
↪comargs[1] == " -Dx" ||  comargs[1] == "-c" ||  comargs[1] == "-C" ||
        comargs[1] == "-config" ||  comargs[1] == "-Config" || comargs[1] == "-f" ||␣
↪comargs[1] == "-h" || comargs[1] == "-H" || comargs[1] == "-F)")
    {
        comargs.erase(comargs.begin() + 1);
        argc --;
    }
    return;
}

vector<string> importConfig(string configfile)
{
    ConfigOption conf;
    conf.setConfig(configfile,true);
    conf.parseConfig();
    vector<string> sections = conf.getSections();
    vector<string> comargs;
    bool maincheck = false;
    for(unsigned i = 0; i < sections.size(); i++)
    {
        if(sections[i] == "main")
        {
            maincheck = true;
            break;
        }
    }
    if(maincheck)
    {
        comargs.push_back(conf.getSectionOptions("main", "job_num"));
        comargs.push_back(conf.getSectionOptions("main", "job_type"));
        comargs.push_back(conf.getSectionOptions("main", "map_config"));
        comargs.push_back(conf.getSectionOptions("main", "output_directory"));
        comargs.push_back(conf.getSectionOptions("main", "min_spec_rate"));
        comargs.push_back(conf.getSectionOptions("main", "zfat"));
        comargs.push_back(conf.getSectionOptions("main", "lval"));
        comargs.push_back(conf.getSectionOptions("main", "deme"));
        comargs.push_back(conf.getSectionOptions("main", "sample_size"));
        comargs.push_back(conf.getSectionOptions("main", "max_time"));
        comargs.push_back(conf.getSectionOptions("main", "lambda"));
```

```
        comargs.push_back(conf.getSectionOptions("main", "time_config"));
        comargs.push_back(conf.getSectionOptions("main", "min_species"));
        if(conf.hasSection("spec_rates"))
        {
            vector<string> spec_rates = conf.getSectionValues("spec_rates");
            for(unsigned i = 0; i < spec_rates.size(); i ++)
            {
                comargs.push_back(spec_rates[i]);
            }
        }
        return(comargs);
    }
    else
    {
        throw Config_Exception("Could not find main section in config file. Aborting
↪");
    }

}

#ifndef infinite_landscape

bool parseArgs(int &argc, vector<string> &comargs)
{
    bool bCheckUser=false;
    if(argc==1)
    {
        comargs.push_back("-e");
        if(comargs.size()!=2)
        {
            cerr << "ERROR_MAIN_010: Incorrect command line parsing." << endl;
        }
    }
    if(comargs[1]=="-h"||comargs[1]=="-H"||argc==1||comargs[1]=="-help" || comargs[1]␣
↪== "-e")
    {
        // Sort out piping to terminal if verbose has not been defined.
        #ifndef verbose
        dup2(saved_stdout, fileno(stdout));
        //close(saved_stdout);
        #endif
        if(argc==1)
        {
            cout << "No arguments supplied: expected 30. These are: " << endl;
        }
        else
        {
            cout << "30 command line arguments are required. These are: " << endl;
        }
        cout << "1: the seed for the simulation." << endl;
        cout << "2: the simulation task (for file reference)." << endl;
        cout << "3: the map config file." << endl;
        cout << "4: the output directory." << endl;
        cout << "5: the minimum speciation rate." << endl;
        cout << "6: the dispersal z_fat value." << endl;
        cout << "7: the dispersal L value." << endl;
        cout << "8: the deme size." << endl;
        cout << "9: the deme sample size." << endl;
```

```
        cout << "10: the maximum simulation time (in seconds)." << endl;
        cout << "11: the lambda value for moving through non-habitat." << endl;
        cout << "12: the temporal sampling file containing tab-separated generation␣
↪values for sampling points in time (null for only sampling the present)." << endl;
        cout << "13: the minimum number of species known to exist. (Currently has no␣
↪effect)." << endl;
        cout << "14 onwards: speciation rates to apply after simulation." << endl;
        cout << "There is also a full-command line mode, (flag -f), which allows for␣
↪more options to be specified via the command line." << endl;
        cout << "Would you like to see these options? Y/N: " << flush;
        string fullopts;
        cin >> fullopts;
        if(fullopts == "Y" || fullopts == "y")
        {
            cout << "1: the task_iter used for setting the seed." << endl;
            cout << "2: the sample grid x dimension." << endl;
            cout << "3: the sample grid y dimension." << endl;
            cout << "4: the fine map file relative path." << endl;
            cout << "5: the fine map x dimension." << endl;
            cout << "6: the fine map y dimension." << endl;
            cout << "7: the fine map x offset." << endl;
            cout << "8 the fine map y offset." << endl;
            cout << "9: the coarse map file relative path." << endl;
            cout << "10: the coarse map x dimension." << endl;
            cout << "11: the coarse map y dimension." << endl;
            cout << "12: the coarse map x offset." << endl;
            cout << "13: the coarse map y offset." << endl;
            cout << "14: the scale of the coarse map compared to the fine (10 means␣
↪resolution of coarse map = 10 x resolution of fine map)." << endl;
            cout << "15: the output directory." << endl;
            cout << "16: the speciation rate." << endl;
            cout << "17: the dispersal distance (zfat)." << endl;
            cout << "18: the deme size." << endl;
            cout << "19: the deme sample size (as a proportion of deme size)." <<␣
↪endl;
            cout << "20: the time to run the simulation (in seconds)." << endl;
            cout << "21: lambda - the relative cost of moving through non-forest." <<␣
↪endl;
            cout << "22: the_task - for referencing the specific task later on." <<␣
↪endl;
            cout << "23: the minimum number of species the system is known to contain.
↪" << endl;
            cout << "24: the pristine fine map file to use." << endl;
            cout << "25: the pristine coarse map file to use." << endl;
            cout << "26: the rate of forest change from pristine." << endl;
            cout << "27: the time (in generations) since the pristine forest was seen.
↪" << endl;
            cout << "28: the dispersal L value (the width of the kernel." << endl;
            cout << "29: the sample mask, with binary 1:0 values for areas that we␣
↪want to sample from. If this is not provided then this will default to mapping the␣
↪entire grid." << endl;
            cout << "30: a file containing a tab-separated list of sample points in␣
↪time (in generations). If this is null then only the present day will be sampled." <
↪< endl;
            cout << "31-onwards: speciation rates to be applied at the end of the␣
↪simulation" << endl;
            cout << "Note that using the -f flag prohibits more than one two historic␣
↪maps being used." << endl;
```

```
        }
        cout << "Would you like to run with the default settings? (Y/N)" << flush;
        string cDef;
        cin >> cDef;
        if(cDef == "Y"||cDef=="y")
        {
            bCheckUser = true;
        }
        else
        {
            bCheckUser = false;
            cout << "Possible command line arguments: " << endl;
            cout << "-h/-help: Show the help file." << endl;
            cout << "-d/-D: Run with default small parameters." << endl;
            cout << "-dl/-DL: Run with default large parameters." << endl;
            cout << "-dx/-DX: Run with the default very large parameters." << endl;
            cout << "-c/-config: Run with the supplied config file." << endl;
            exit(1); // exit the program right away as there is no need to continue␣
↪if there is no simulation to run!
        }
        #ifndef verbose
        openLogFile(true);
        #endif
    }
    // Import the default parameters if required.
    if(comargs[1]=="-d"||comargs[1]=="-D"||bCheckUser)
    {
        runAsDefault(comargs);
        bCheckUser=true;
    }
    if(comargs[1]=="-dl"||comargs[1]=="-DL"||comargs[1]=="-dL"||comargs[1]=="-Dl")
    {
        runLarge(comargs);
        bCheckUser = true;
    }
    if(comargs[1]=="-dx"||comargs[1]=="-dX"||comargs[1]=="-DX"||comargs[1]=="-Dx")
    {
        runXL(comargs);
        bCheckUser = true;
    }
    bool bConfig = false;
    if(comargs[1]=="-c"||comargs[1]=="-C"||comargs[1]=="-config"|| comargs[1]=="-
↪Config")
    {
        // Check that the config file is supplied.
        if(argc!=3)
        {
            throw Main_Exception("ERROR_MAIN_011: FATAL. -c or -config used to␣
↪attempt import from config file, but no config file provided.");
        }
        bConfig = true;
        cout << "Importing values from config file " << comargs[2] << "...." << flush;
        try
        {
            vector<string> tmp_comargs = importConfig(comargs[2]);
            vector<string> tmp2_comargs;
            tmp2_comargs.push_back(comargs[0]);
            tmp2_comargs.push_back(comargs[1]);
```

```
            tmp2_comargs.insert(tmp2_comargs.end(), tmp_comargs.begin(), tmp_comargs.
↪end());
            comargs = tmp2_comargs;
//          for(unsigned i = 0; i < tmp2_comargs.size(); i++)
//          {
//              cout << tmp2_comargs[i] << endl;
//          }
            argc = comargs.size();
//          argc = conf.importConfig(comargs);
        }
        catch(Config_Exception& c_e)
        {
            cerr << c_e.what() << endl;
            throw Fatal_Exception();
        }
        //cout << "test" << endl;
        //cout << endl << comargs[0] << endl;
        cout << "done!" << endl;
    }
    bool bFullMode = false;
    if(comargs[1] == "-f" || comargs[2] == "-f")
    {
        cout << "Full command-line mode enabled." << endl;
        bFullMode = true;
    }
    removeComOption(argc, comargs);
    removeComOption(argc, comargs);
    if(argc > 12 && !bFullMode)
    {
        return(false);
    }
    if(argc<31&&!bCheckUser &&!bConfig)
    {
        string err = "ERROR_MAIN_000: FATAL.  Incorrect arguments supplied (" + to_
↪string((long long)argc-1) + " supplied; expected 30).";
        throw Main_Exception(err);
        // note argc-1 which takes in to account the automatic generation of one␣
↪command line argument which is the number of arguments.
    }
    argc = comargs.size();
    return(true);
}

#endif

#ifdef infinite_landscape

bool parseArgs(int &argc, vector<string> &comargs)
{
    //bool bCheckUser=false;
    if(argc==1)
    {
        comargs.push_back("-e");
        if(comargs.size()!=2)
        {
            cerr << "ERROR_MAIN_010: Incorrect command line parsing." << endl;
        }
    }
```

```
    bool rundef = false;
    if(comargs[1]=="-h"||comargs[1]=="-H"||argc==1||comargs[1]=="-help" || comargs[1]␣
↪== "-e")
    {
        #ifndef verbose
        dup2(saved_stdout, fileno(stdout));
        //close(saved_stdout);
        #endif
        if(argc==1)
        {
            cout << "No arguments supplied: expected 30. These are: " << endl;
        }
        else
        {
            cout << "6 command line arguments are required. These are: " << endl;
        }
        cout << "1: the seed for the simulation." << endl;
        cout << "2: the simulation task (for file reference)." << endl;
        cout << "3: the output directory." << endl;
        cout << "4: the minimum speciation rate." << endl;
        cout << "5: the dispersal sigma value." << endl;
        cout << "6: the size (width and length) of the simulation." << endl;
        cout << "Would you like to run with default variables? Y/N: " << flush;
        string user_default;
        cin >> user_default;
        if(user_default == "Y" || user_default == "y" )
        {
            rundef = true;
            comargs[1] = "1";
            comargs.push_back("0");
            comargs.push_back("Default");
            comargs.push_back("0.01");
            comargs.push_back("4");
            comargs.push_back("100");
        }
        else
        {
            cout << "Nothing to do, exiting." << endl;
            exit(0);
        }
        #ifndef verbose
        openLogFile(true);
        //close(saved_stdout);
        #endif
    }
    if( comargs[1] == "-d" || comargs[1] == "-D" || comargs[1] == "-DX" || comargs[1]␣
↪== "-Dx" || comargs[1] == "-dX" || comargs[1] == "-dx" ||comargs[1]=="-dl
↪"||comargs[1]=="-DL"||comargs[1]=="-dL"||comargs[1]=="-Dl")
    {
        vector<string> newcomargs;
        runAsDefault(newcomargs);
        comargs = newcomargs;
    }
    if( comargs.size() == 7)
    {
        vector<string> newcomargs;
        runAsDefault(newcomargs);
        newcomargs[1] = comargs[1];
```

```
        newcomargs[2] = comargs[6];
        newcomargs[3] = comargs[6];
        newcomargs[5] = comargs[6];
        newcomargs[6] = comargs[6];
        newcomargs[7] = "0.0";
        newcomargs[8] = "0.0";
        newcomargs[10] = comargs[6];
        newcomargs[11] = comargs[6];
        newcomargs[12] = "0.0";
        newcomargs[13] = "0.0";
        newcomargs[14] = "1";
        newcomargs[15] = comargs[3];
        newcomargs[16] = comargs[4];
        newcomargs[17] = comargs[5];
        newcomargs[18] = "1.0";
        newcomargs[19] = "1.0";
        newcomargs[20] = "172800";
        newcomargs[22] = comargs[2];
        newcomargs.push_back(comargs[4]);
        string firstcom = comargs[0];
        comargs.clear();
        comargs.push_back(firstcom);
        comargs.insert(comargs.end(), newcomargs.begin(), newcomargs.end());
        //cout << "argc: '" << argc << endl;
        argc = comargs.size();
        //cout << "argc: '" << argc << endl;
    }

    bool bFullMode = false;
    if(comargs[1] == "-f" || comargs[2] == "-f")
    {
        cout << "Full command-line mode enabled." << endl;
        bFullMode = true;
    }
    removeComOption(argc, comargs);
    removeComOption(argc, comargs);

    if(argc > 12 && !bFullMode)
    {
        return(false);
    }
    return(true);
}

#endif

void getSR(int argc,const  vector<string> & comargs, Tree & t)
{
    if(argc>31)
    {
        unsigned int iMultiNumber = argc - 31;
        vector<double> dSpecArray;
        for(int i =31;i<argc;i++)
        {
            dSpecArray.push_back(stod(comargs[i]));
        }
        cout << "Speciation rate" << flush;
        if(dSpecArray.size() > 1)
```

```
        {
            cout << "s are: " << flush;
        }
        else
        {
            cout << " is: " << flush;
        }
        for(unsigned int i=0;i<iMultiNumber;i++)
        {
            cout << dSpecArray[i] << flush;
            if(i +1 == iMultiNumber)
            {
                cout << "." << endl;
            }
            else
            {
                cout << ", " << flush;
            }
        }
        // Now check to make sure repeat speciation rates aren't done twice (this is␣
↪done to avoid the huge number of errors SQL throws if you try to add identical data
        double dUniqueSpec[iMultiNumber];
        unsigned long spec_upto = t.sortData();
        t.sqlCreate();
        for(unsigned int i =0;i<iMultiNumber;i++)
        {
            bool bCont = true;
            for(unsigned int j =0;j<iMultiNumber;j++)
            {
                if(dUniqueSpec[j]==dSpecArray[i])
                {
                    bCont = false;
                }
            }
            if(bCont)
            {
                vector<double> temp_sampling = t.getTemporalSampling();
                for(unsigned k = 0; k < temp_sampling.size(); k++)
                {
                    try
                    {
                        cout << "Calculating generation " << temp_sampling[k] << endl;
                        t.applySpecRates(dSpecArray[i], temp_sampling[k]);
                    }
                    catch(const std::exception & e)
                    {
                        cout << e.what() << endl;
                    }
                }
            }
            else
            {
                cout << "Repeat speciation rate... ignoring" << endl;
            }
        }
        t.outputData(spec_upto);
    }
    else
```

```
    {
        t.sqlCreate();
        t.outputData();
        return;
    }
}


bool doesExist(string testfile)
{
    if(boost::filesystem::exists(testfile))
    {
            cout << "\rChecking folder existance..." << testfile << " exists!
↪       " << flush;
        return true;
    }
    else
    {
        cout << testfile << " does not exist!                " << endl;
        throw runtime_error(string("ERROR_MAIN_008: FATAL. Input or output folder
↪does not exist: " + testfile + "."));
    }
    return false;
}

bool doesExistNull(string testfile)
{
    if(testfile=="null")
    {
        return(true);
    }
    else
    {
        return(doesExist(testfile));
    }
}
```

Includes

- `Setup.h` (*File Setup.h*)

Functions

- *Function charconvertor*
- *Function checkFolders*
- *Function checkSims*
- *Function doesExist*
- *Function doesExistNull*
- *Function getSR*
- *Function importConfig*

- *Function jobConvertor*
- *Function openLogFile*
- *Function parseArgs*
- *Function removeComOption*
- *Function runAsDefault*
- *Function runLarge*
- *Function runXL*

Variables

- *Variable log_name*
- *Variable saved_stdout*

File Setup.h

Definition (`Setup.h`)

Program Listing for File Setup.h

- Return to documentation for *File Setup.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
//
#ifndef SETUP
#define SETUP
#include <string>
#include <vector>
#include <unistd.h>
#include <sstream>
#include <ctime>
#include <boost/filesystem.hpp>
#include <stdio.h>
#include <time.h>
#include <iostream>
#include <iomanip>
#include "Tree.h"

// Forward declaring the global variables
// store the log file name for access anywhere.
extern string log_name;
// the old stdout
extern int saved_stdout;
using namespace std;
long charconvertor(char charin);

long jobConvertor(char * argin);

void checkFolders(string sFineMap, string sCoarseMap, string sFineMapPristine, string␣
↪sCoarseMapPristine, string sOutputFolder, string sSampleMask);
```

```
bool checkSims(int the_task, int the_seed, string outdirect);

void getSR(int argc, const vector<string>&comargs,Tree&t);
#ifndef verbose
void openLogFile(bool append);
#endif
bool parseArgs(int& argc, vector<string>& comargs);

void removeComOption(int& argc, vector<string> & comargs);

void runAsDefault(vector<string>&comargs);

void runLarge(vector<string>&comargs);

void runXL(vector<string>&comargs);

bool doesExist(string testfile);

bool doesExistNull(string testfile);

#endif // SETUP
```

Includes

- Tree.h (*File Tree.h*)
- boost/filesystem.hpp
- ctime
- iomanip
- iostream
- sstream
- stdio.h
- string
- time.h
- unistd.h
- vector

Included By

- *File main.cpp*
- *File Map.cpp*
- *File Setup.cpp*
- *File SpeciationCounter.cpp*

Functions

- *Function charconvertor*
- *Function checkFolders*
- *Function checkSims*
- *Function doesExist*
- *Function doesExistNull*
- *Function getSR*
- *Function jobConvertor*
- *Function openLogFile*
- *Function parseArgs*
- *Function removeComOption*
- *Function runAsDefault*
- *Function runLarge*
- *Function runXL*

Variables

- *Variable log_name*
- *Variable saved_stdout*

File SpeciationCounter.cpp

Definition (`SpeciationCounter.cpp`)

Program Listing for File SpeciationCounter.cpp

- Return to documentation for *File SpeciationCounter.cpp*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.

#include <stdio.h>
//#include "fast-cpp-csv-parser/csv.h"
#include "Treelist.h"
#include "Config.h" // Just for the importing of command-line arguments for now.␣
↪Later on parsing parameters from a file might be implemented.
#include "Setup.h"
#include <boost/filesystem.hpp>


using namespace std;
// INPUTS
// requires a SQL database file containing the the Treenode objects from a␣
↪coalescence simulations.
```

```
// the required speciation rate.

// OUTPUTS
// An updated database file that contains the species richness and species abundances␣
↪of the intended lineage.

struct simParameters
{
    bool RecordSpatial;
    bool bMultiRun;
    bool RecordFragments;
    string filename;
    vector<double> vSpecRates;
    string samplemask;
    string autocorrel_file;
    vector<double> autocorrel_times;
    bool bAuto;
    string fragment_config_file;
};

void parseArgs(int &argc, vector<string> &comargs, simParameters &sp)
{
    bool bRunDefault=false;
    bool bInvalidArguments = false;
    bool bAskHelp = false;
    #ifndef verbose
    dup2(saved_stdout, fileno(stdout));
    //close(saved_stdout);
    #endif
    if(argc<7)
    {
        if(argc == 1)
        {
            bInvalidArguments = true;
        }
        bInvalidArguments = true;
        if(argc==2)
        {
            if((comargs[1])=="-d")
            {
                bInvalidArguments = false;
                bRunDefault = true;
            }
            if(comargs[1]=="-h"||comargs[1]=="-help")
            {
                bInvalidArguments = false;
                bAskHelp = true;
            }
        }
        if(bInvalidArguments)
        {
            cout << "Incorrect number of arguments." << endl;
            bInvalidArguments = true;
            if(argc == 1)
            {
                comargs.push_back("-e");
            }
            else
```

```
            {
                comargs[1] = "-e";
            }
        }
    }
    else
    {
        sp.samplemask = comargs[3];
        sp.filename = comargs[1];
        sp.autocorrel_file = comargs[4];
    }
    if(argc>7)
    {
        sp.bMultiRun = true;
        int i = 6;
        while(i<argc)
        {
            sp.vSpecRates.push_back(stof(comargs[i]));
            i++;
        }
    }
    else if(argc == 7 && !bInvalidArguments && !bAskHelp)
    {
        sp.bMultiRun = false;
        sp.vSpecRates.push_back(stod(comargs[6]));
    }
    if(!bInvalidArguments && !bAskHelp && !bRunDefault)
    {
        if(comargs[2] == "true" || comargs[2] == "True" || comargs[2] == "T" ||␣
↪comargs[2] == "TRUE" || comargs[2] == "t")
        {
            sp.RecordSpatial = true;
        }
        else
        {
            sp.RecordSpatial = false;
        }
        if(comargs[5] == "false" || comargs[5] == "False" || comargs[5] == "F" ||␣
↪comargs[5] == "FALSE" || comargs[5] == "f")
        {
            sp.RecordFragments = false;
        }
        else
        {
            if(comargs[5] == "true" || comargs[5] == "True" || comargs[5] == "T" ||␣
↪comargs[5] == "TRUE" || comargs[5] == "t")
            {
                sp.fragment_config_file = "null";
            }
            else
            {
                sp.fragment_config_file = comargs[5];
            }
            sp.RecordFragments = true;
        }
    }
    if(bInvalidArguments||bAskHelp)
    {
```

```
        cout << "At least six command-line arguments are expected." << endl;
        cout << "1 - Path to SQL database file." << endl;
        cout << "2 - T/F of whether to record full spatial data." << endl;
        cout << "3 - the sample mask to use (use null if no mask is to be used)" <<␣
↪endl;
        cout << "4 - the file containing tempororal points of interest. If null, the␣
↪present is used for all calculations." << endl;
        cout << "5 - T/F of whether to calculate abundances for each rectangular␣
↪fragment. Alternatively, provide a csv file with fragment data to be read." << endl;
        cout << "6 - Speciation rate." << endl;
        cout <<"7 - onwards - Further speciation rates. [OPTIONAL]" << endl;
        cout << "Would you like to run with the default paramenters?" << endl;
        cout << "       (This requires a SQL database file at ../../Data/Coal_sim/
↪Test_output/SQL_data/data_0_1.db)" << endl;
        cout << "Enter Y/N: " << flush;
        string sDef;
        cin >> sDef;
        if(sDef=="Y"||sDef=="y")
        {
            bRunDefault = true;
        }
        else
        {
            bRunDefault = false;
            exit(0);
        }
    }
    if(comargs[1]=="-d" || bRunDefault)
    {
        sp.filename= "../../Data/Coal_sim/Test_output/SQL_data/data_0_1.db";
        sp.vSpecRates.push_back(0.001);
        sp.samplemask = "null";
        sp.autocorrel_file = "null";
        sp.fragment_config_file = "null";
        sp.RecordFragments = false;
        sp.RecordSpatial = true;
    }
    try
    {
        if(sp.autocorrel_file == "null")
        {
            sp.bAuto = false;
        }
        else
        {
            sp.bAuto = true;
            vector<string> tmpimport;
            ConfigOption tmpconfig;
            tmpconfig.setConfig(sp.autocorrel_file,false);
            tmpconfig.importConfig(tmpimport);
            for(unsigned int i = 0;i<tmpimport.size();i++)
            {
                sp.autocorrel_times.push_back(stod(tmpimport[i]));
//                  cout << "t_i: " << sp.autocorrel_times[i] << endl;
            }
        }
    }
    catch(Config_Exception& ce)
```

```
    {
        cerr << ce.what() << endl;
    }
    #ifndef verbose
    openLogFile(true);
    #endif
}



int main(int argc, char **argv)
{
    time_t tStart,tEnd;
    time(&tStart);
    simParameters sp;
    Row<Treenode> data;
//  sqlite3 * database;
    string inputfile;
    unsigned int iMultiNumber = argc-6;
    #ifndef verbose
    openLogFile(false);
    #endif
    // vector to store the command-line arguments
    vector<string> comargs;
    importArgs(argc,argv,comargs);
    parseArgs(argc,comargs,sp);
    // Import the arguments
//  for(int i =0;i < comargs.size();i++)
//  {
//      cout << comargs[i] << endl;
//  }


    // Now do the actual calculation
    // First print the variables
    cout << "**************************" << endl;
    cout << "STARTING CALCULATIONS" << endl;
    cout << "Input file is " << sp.filename << endl;
    if(!sp.bMultiRun)
    {
        cout << "Speciation rate is " << sp.vSpecRates[0] << endl;
    }
    else
    {
        cout << "Speciation rates are: " << flush;
        for(unsigned int i=0;i<iMultiNumber;i++)
        {
            cout << sp.vSpecRates[i] << flush;
            if(i +1 == iMultiNumber)
            {
                cout << "." << endl;
            }
            else
            {
                cout << ", " << flush;
            }
        }
    }
```

```
    // Set up the objects
    Treelist nodes(data);
    nodes.setList(data);
//  unsigned long iCount;
    // Import the data from the file into the Row<Treenode> object.
    try
    {
//      nodes.detectDimensions(sp.filename);
        nodes.importSamplemask(sp.samplemask);
        nodes.importData(sp.filename);
        if( sp.RecordFragments)
        {
            nodes.calcFragments(sp.fragment_config_file);
        }
    }
    catch(exception& se)
    {
        cerr << se.what() << endl;
        exit(-1);
    }
    // Get rid of any previous speciation calculations
    nodes.resetTree();
    nodes.setGeneration(0);
    // Calculate the new tree structure.
    if(sp.bMultiRun)
    {
        vector<double> dUniqueSpec;
        for(unsigned int i =0;i<iMultiNumber;i++)
        {
            bool bCont = true;
            for(unsigned int j =0;j<dUniqueSpec.size();j++)
            {
                if(dUniqueSpec[j]==sp.vSpecRates[i])
                {
                    bCont = false;
                }
            }
            if(bCont)
            {
                if(sp.bAuto)
                {
                    dUniqueSpec.push_back(sp.vSpecRates[i]);
                    for(unsigned int k = 0;k < sp.autocorrel_times.size(); k++)
                    {
                        cout << "Calculating generation " << sp.autocorrel_times[k] <
↪< endl;
                        nodes.setGeneration(sp.autocorrel_times[k]);
                        nodes.resetTree();
                        try
                        {
                            nodes.createDatabase(sp.vSpecRates[i]);
                            if(sp.RecordSpatial)
                            {
                                nodes.recordSpatial();
                            }
                            if(sp.RecordFragments)
```

```
                            {
                                nodes.applyFragments();
                            }
                        }
                        catch(SpeciesException &se)
                        {
                            cerr << se.what() << endl;
                        }
                    }
                }
                else
                {
                    dUniqueSpec.push_back(sp.vSpecRates[i]);
                    nodes.resetTree();
                    try
                    {
                        nodes.createDatabase(sp.vSpecRates[i]);
                        if(sp.RecordSpatial)
                        {
                            nodes.recordSpatial();
                        }
                        if(sp.RecordFragments)
                        {
                            nodes.applyFragments();
                        }

                    }
                    catch(SpeciesException &se)
                    {
                        cerr << se.what() << endl;
                    }
                }
            }
            else
            {
                cout << "Repeat speciation rate... ignoring" << endl;
            }
        }
    }
    else
    {
        if(sp.bAuto)
        {
            for(unsigned int k = 0;k < sp.autocorrel_times.size(); k++)
            {
                cout << "Calculating generation " << sp.autocorrel_times[k] << endl;
                nodes.setGeneration(sp.autocorrel_times[k]);
                nodes.resetTree();
                try
                {
                    nodes.createDatabase(sp.vSpecRates[0]);
                    if(sp.RecordSpatial)
                    {
                        nodes.recordSpatial();
                    }
                    if(sp.RecordFragments)
                    {
                        nodes.applyFragments();
```

```
                    }
                }
                catch(SpeciesException &se)
                {
                    cerr << se.what() <<endl;
                }
            }
        }
        else
        {
            try
            {
                nodes.createDatabase(sp.vSpecRates[0]);
                if(sp.RecordSpatial)
                {
                    nodes.recordSpatial();
                }
                if(sp.RecordFragments)
                {
                    nodes.applyFragments();
                }
            }
            catch(SpeciesException &se)
            {
                cerr << se.what() << endl;
            }
        }
    }
    nodes.exportDatabase(sp.filename);
//  cout << "speciation rate of 0.0001: " << list.calcSpecies(0.0001);
//  list.resetTree();
//  cout << "speciation rate of 0.001: " << list.calcSpecies(0.001);
//  list.resetTree();
//  cout << "speciation rate of 0.01: " << list.calcSpecies(0.01);
    time(&tEnd);
    cout << "Calculations complete." << endl;
    cout << "Time taken was " <<floor((tEnd - tStart)/3600) << " hours " <<␣
↪(floor((tEnd - tStart)/60)-60*floor((tEnd - tStart)/3600)) << " minutes " << (tEnd-
↪tStart)%60<< " seconds" << endl;

    return 0;
}
```

Includes

- `Config.h` (*File Config.h*)
- `Setup.h` (*File Setup.h*)
- `Treelist.h` (*File Treelist.h*)
- `boost/filesystem.hpp`
- `stdio.h`

Classes

- *Class simParameters*

Functions

- *Function main*
- *Function parseArgs*

File SpeciesList.h

Definition (`SpeciesList.h`)

Program Listing for File SpeciesList.h

- Return to documentation for *File SpeciesList.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.

/*****************************************************************
                    SPECIES LIST OBJECT
 *****************************************************************/
 #ifndef SPECIESLIST
 #define SPECIESLIST
 #include <iostream>
using namespace std;
class SpeciesList
{
private:
    unsigned short list_size,maxsize; // List size and maximum size of the cell␣
↪(based on percentage cover).
    unsigned long next_active; // For calculating the wrapping, using the next and␣
↪last system.
    Row<unsigned long> list; // list of the active reference number, with zeros for␣
↪empty cells.
    unsigned short nwrap; // The number of wrapping (next and last possibilities)␣
↪that there are.
public:
    SpeciesList() : list_size(0),maxsize(0),next_active(0),nwrap(0)
    {
        list.SetRowSize(0);
    }

    ~SpeciesList()
    {

    }
    // Sets the list size to the required length.
    // Note this will delete any species currently stored in the list

    // Fill the list with empty 0s.
```

```
    void fillList()
    {
        if(maxsize==0)
        {
            return;
        }
        for(unsigned int i = 0;i<maxsize;i++)
        {
            list[i] = 0;
        }
    }

    // Standard setters
    void initialise(unsigned short maxsizein)
    {
        maxsize = maxsizein;
        nwrap = 0;
        list_size = 0;
        list.SetRowSize(maxsize);
        return;
    }

    // special case if just the maxsize wants to be change, but want to maintain the␣
↪list variables.
    void setMaxsize(unsigned short maxsizein)
    {
        maxsize = maxsizein;
        return;
    }

    void setSpecies(int index, unsigned long new_val)
    {
        if(list[index] == 0)
        {
            cerr << "index: " << index << endl;
            cerr << "list[index]: " <<list[index] << endl;
            cerr << "list.maxsize(): " << maxsize << endl;
            throw runtime_error("ERROR_MOVE_027: List position to be replaced is zero.
↪ Check list assignment.");
        }
        list[index] = new_val;
        return;
    }

    void setNext(unsigned long n)
    {
        next_active = n;
    }


    void setNwrap(unsigned short nr)
    {
        nwrap = nr;
    }

    unsigned short addSpecies(unsigned long new_spec)
    {
        /*
```

```
        list_size ++;
        if(list_size>maxsize)
        {
            list_size --;
            return;
        }
        else
        {
            list[list_size] = new_spec;
        }
         * */
        // Alternative method
        // loop until an empty space is found and place the new species in the empty␣
↪space.
        if(list_size + 1 > maxsize)
        {
            cerr << "maxsize: " << maxsize << endl;
            throw out_of_range("Could not add species - no empty space");
        }
        for(unsigned short i =0; i<list.size(); i++)
        {
            if(list[i] == 0)
            {
                list_size++;
                list[i] = new_spec;
                return i;
            }

        }
        cerr << "maxsize: " << maxsize << endl;
        throw out_of_range("Could not add species - no empty space");
    }

    void addSpeciesSilent(unsigned long new_spec)
    {
        /*
        list_size ++;
        if(list_size>maxsize)
        {
            list_size --;
            return;
        }
        else
        {
            list[list_size] = new_spec;
        }
         * */
        // Alternative method
        // loop until an empty space is found and place the new species in the empty␣
↪space.
        for(unsigned int i =0;i<maxsize;i++)
        {
            if(list[i] == 0)
            {
                list_size++;
                list[i] = new_spec;
                return;
            }
```

```
        }
        throw out_of_range("Could not add species - no empty space");
    }

    void deleteSpecies(int index)
    {
//      cout << "deleted species index: " << index << " value: " << list[index] << "␣
↪list size: " << list_size << endl;
        list[index] = 0;
        list_size --;

        // alternative version
        /*for(int i = index;i<list_size;i++)
        {
            list[i] = list[i+1];
        }
        list[list_size]=0;
        list_size --;*/
    }

    void decreaseNwrap()
    {
        if(nwrap == 0)
        {
            throw runtime_error("Nwrap should never be decreased less than 0");
        }
        else if(nwrap == 1)
        {
            if( next_active != 0)
            {
                throw runtime_error("Nwrap is being set at 0 when an wrapped lineage␣
↪is still present");
            }
        }
        nwrap --;
    }

    void increaseNwrap()
    {
        nwrap ++;
    }

    void changePercentCover(unsigned short newmaxsize) // THIS NEEDS TO BE CHANGED TO␣
↪TAKE ACCOUNT OF THE NEW REFERENCING METHOD
    {
//      cout << "1" << endl;
        Row<unsigned long> templist(list);
        maxsize = newmaxsize;
        list.SetRowSize(newmaxsize);
//      cout << "2" << endl;
//      cout << "templist.size: " << templist.size() << endl;
//      cout << "list.size: " << list.size() << endl;
//      cout << "list_size: " << list_size << endl;
//      cout << "maxsize: " << maxsize << endl;
//      cout << "newmaxsize: " << newmaxsize << endl;
//      cout << "templist[0]: " << templist[0] << endl;
        for(unsigned int i=0;i<newmaxsize;i++)
```

```
        {
//          cout << i << endl;
            if(i<templist.size()) // DOUBLE CHECK THIS LATER
            {
//              cout << i << endl;
                list[i] = templist[i];
//              cout << "assignment done" << endl;
            }
            else
            {
//              cout << "alt: " << i << endl;
                list[i] = 0;
            }
        }
        if(list.size() > maxsize)
        {
            cerr << "list.size(): " << list.size() << endl;
            cerr << "maxsize: " << maxsize << endl;
            throw out_of_range("List size not equal to maxsize");
        }
//      cout << "3" << endl;
        // Alternative method
//      maxsize = newmaxsize;
    }

    unsigned long getRandLineage(NRrand &rand_no)
    {
        double rand_index;
        if(maxsize <= list_size)
        {
            // Then the list size is larger than the actual size. This means we must␣
↪return a lineage.
            try
            {
                do
                {
                    rand_index = rand_no.d01();
                    rand_index *= list.size();
                    //cout << "ref: " << rand_index << ", " << list[round(rand_
↪index)] << endl;
                } while(list[floor(rand_index)] == 0);
                //cout << "RETURNING!" << endl;
                return(list[floor(rand_index)]);
            }
            catch(out_of_range &oor)
            {
                cerr << oor.what() << endl;
                throw runtime_error("ERROR_MOVE_001b: Listpos outside maxsize.");
            }
        }
        else
        {
            rand_index =  rand_no.d01();
//      cout << "rand_index: " << rand_index << endl;
            rand_index  *= maxsize;
        // Dynamically resize the list if required. Otherwise, to save memory, the␣
↪list will not be resized;
            if(rand_index>=list.size())
```

```
            {
    //          cout << "changing: " << endl;
                changePercentCover(maxsize);
    //          string sam;
    //          cin >> sam;
            }
            #ifdef DEBUG
//          if(list.size() > 1 || list_size > 1 || maxsize != 1)
//          {
//              throw runtime_error("HELP!");
//          }
            #endif
            // now get the correctly placed lineage (returning 0s when necessary)
            /*int j =0; // this counts the number of 0s we've seen (included)
            int i = 0; // this counts the total number we've passed
            int k =0; // this counts the  number of 0s seen (skipped)
            int zeroes = maxsize - list_size;
            while(i-k < rand_index)
            {
                i++;
                // then we need to return a lineage
                if(list[i] == 0)
                {
                    if(j < zeroes)
                    {
                        j ++;
                    }
                    else
                    {
                        k++;
                    }
                }
            }*/

            // test new version
            unsigned int i = floor(rand_index);

            // REMOVE THIS LATER
            if(rand_index>maxsize)
            {
                cout << "ERROR IN INDEX" << endl;
                string sam;
                cin >> sam;
            }
            return list[i];
        }
    }

    unsigned long getSpecies(unsigned short index)
    {
        return list[index];
    }

    unsigned long getNext()
    {
        return next_active;
    }
```

```
    unsigned short getNwrap()
    {
        return nwrap;
    }

    int getListsize()
    {
        return list_size;
    }

    int getMaxsize()
    {
        return maxsize;
    }

    void wipeList()
    {
        fillList();
        next_active=0;
        nwrap =0;
        list_size=0;
    }

    friend ostream& operator<<(ostream& os,const SpeciesList& r)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        os << r.list << ",";
        os << r.list_size << ",";
        os << r.maxsize << ",";
        os << r.next_active << ",";
        os << r.nwrap << ",";
        return os;
    }

    friend istream& operator>>(istream& is, SpeciesList& r)
    {
        char delim;
        //double temp1,temp2;
        //is << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        is >> r.list;
        is >> delim;
        is >> r.list_size;
        is >> delim;
        is >> r.maxsize;
        is >> delim;
        is >> r.next_active;
        is >> delim;
        is >> r.nwrap;
        is >> delim;
        return is;
    }
};

#endif
```

Includes

- iostream

Included By

- *File Tree.h*

Classes

- *Class SpeciesList*

File Tree.h

Definition (`Tree.h`)

Program Listing for File Tree.h

- Return to documentation for *File Tree.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
//
#ifndef TREE
#define TREE


/*************************************************************
                        INCLUDES
 *************************************************************/
// standard includes
# include <stdio.h>
# include <fstream>
# include <vector>
# include <iostream>
# include <string>
# include <cstring>
# include <math.h>
# include <iomanip>
# include <cmath>
# include <time.h>
# include <ctime>
# include <sqlite3.h>
#include <string>
//# include <sqlite.h>
# include <unistd.h>
# include <algorithm>
# include <stdexcept>
//#define with_gdal
// extra boost include - this requires the installation of boost on the system
// note that this requires compilation with the -lboost_filesystem and -lboost_system␣
↪linkers.
```

```
# include <boost/filesystem.hpp>

// include fast-csv-parser by Ben Strasser (available from https://github.com/ben-
↪strasser/fast-cpp-csv-parser)
// for fast file reading
#ifdef use_csv
#include "fast-cpp-csv-parser/csv.h"
#endif

//#define use_csv // for integration with the matrix header file
//#include "fast-cpp-csv-parser/csv.h"
//#define record_space // tells the compiler whether to include the routines for␣
↪outputting full spatial data of lineages. Usually this will not be required.

// this uses the RAM for the storage of the active SQL database.
// If the RAM requirements get too huge, comment this out to instead write directly␣
↪to disc.
// For HPC systems, it is recommended to use this option as write speeds are␣
↪generally fast and large simulations don't
// have a linear increase in the SQL database size (at least in RAM).
#define sql_ram


// other includes for required files
# include "Matrix.h"
# include "Fattaildeviate.h"
# include "Datapoint.h"
# include "Treenode.h"
# include "SpeciesList.h"
# include "Map.h"
# include "Treelist.h"
# include "Config.h"


using namespace std;

/*************************************************************
                        MAIN TREE OBJECT
 *************************************************************/

class Tree
{

// declare private variables
private:
    // storing the coalescence tree itself
    Row<Treenode> data;
    unsigned long enddata;
    // for storing the command line parameters and parsing the required information.
    Mapvars mapvarsin;
    // random number generator
    NRrand NR;
    bool seeded;
    // random seed
    long long the_seed;

    // for general debugging use
    bool debug;
```

```
    // for enabling the logging mode
    bool log_all;
    // note: in earlier versions I had the minspecsetup variable here
    // I've removed it because this version implements speciation as it goes rather␣
↪than on the tree later
    // I've also removed sim_counter which recorded the number of repeat simulations
    // because each job will be 1 simulation in this implementation

    // for file naming - good to know which task in a series is being executed here
    long long the_task;

    // The map file containing the times that we want to expand the model and record␣
↪all lineages again.
    // If this is null, bAutocorrel will be false and the vector will be empty.
    string autocorrel_file;
    vector<double> autocorrel_times;
    // Set to true if we are recording at times other than the present day.
    bool bAutocorrel;
    // the map files // these are now removed from the class object and replaced with␣
↪a function fed by a series of strings serving the same purpose.
    /*vector<string> map_list;
    // the sizes of the same map files
    vector<long> map_x_size;
    vector<long> map_y_size;
    // The list of size of the grid within the map and the dimensions of the offset.
    vector<long> grid_x_size;
    vector<long>grid_y_size;
    vector<long> grid_x_offset;
    vector<long>grid_y_offset;
     * */
     //tmp debugging
     bool bSpec;
    // A list of new variables which will contain the relevant information for maps␣
↪and grids.
    string finemapinput, coarsemapinput, outdirectory; //  strings containing the␣
↪file names to be imported.
    string pristinefinemapinput, pristinecoarsemapinput;
    double dPristine, dForestTransform; // the time since pristine forest and the␣
↪rate of change of the rainforest.
    long gridxsize, gridysize; // the variables for the grid containing the initial␣
↪individuals.
    long finemapxsize, finemapysize, finemapxoffset, finemapyoffset; // The fine map␣
↪variables at the same resolution as the grid.
    long coarsemapxsize, coarsemapysize, coarsemapxoffset, coarsemapyoffset,
↪coarsemapscale; // the coarse map variables at a scaled resolution of the fine map.
    bool varimport; // Used to check whether the map variables have already been␣
↪imported.
    // New private vectors
    // The time variables (for timing the simulation in real time)
    time_t start, sim_start,sim_end,now,tCheckSpec,sim_finish, out_finish;
    // Map object containing both the coarse and fine maps for checking whether or␣
↪not there is forest at a particular location.
    Map forestmap;
    // An indexing spread for the lineages
    Matrix<SpeciesList> grid;
    Row<Datapoint> active;
    // Active lineages stored as a row of datapoints
```

```
    // Stores the point of the end of the active vector
    unsigned long endactive; // 0 is reserved as null
    unsigned long startendactive; // the maximum size of endactive
    unsigned long maxsimsize; // the maximum simulated number of individuals in the␣
↪present day.
    //This might need to be updated for simulations that have large changes in␣
↪maximum population size over time.

    // number of simulation steps
    long steps;
    // number of generations passed, dispersal and zfat references
    double generation, zfat, dLvalue, maxtime, deme_sample;
    long double spec;
    double lambda; // the cost for moving through non-forest. 1.0 means there is no␣
↪cost. 10 means that movement is 10x slower through forest.
    long deme;
    unsigned long desired_specnum; // the desired number of species we are aiming for.
↪ If it is 0, we will carry on forever.
    //sqlite3 object that stores all the data
    sqlite3 *database;
    // If sql database is written first to memory, then need another object to␣
↪contain the in-memory database.
    #ifdef sql_ram
    sqlite3 *outdatabase;
    #endif
    string sqloutname;
    Treelist tl; // for create the link to the speciationcounter object which handles␣
↪everything.
    Datamask samplegrid; // contains the Datamask for where we should start lineages␣
↪from.
public:

    // constructor
    Tree() : tl(data)
    {
        enddata =0;
        seeded = false;
        the_seed = -10;
        debug = false; // set this equal to true if you want to enable debugging
        log_all = true; //set this equal to true if you want to log every 5 seconds␣
↪to a logfile.
        the_task = -1;
        varimport = false;
        sqloutname = "null";
    }


    void setMapVars(vector<string> comargs, bool fullmode)
    {
        mapvarsin.import(comargs, fullmode);
        mapvarsin.printVars();
        if(!varimport)
        {

            // Set the variables equal to the value from the Mapvars object.
            finemapinput = mapvarsin.finemapfile;
            coarsemapinput = mapvarsin.coarsemapfile;
            gridxsize = mapvarsin.vargridxsize;
```

```
            gridysize = mapvarsin.vargridysize;

            finemapxsize = mapvarsin.varfinemapxsize;
            finemapysize = mapvarsin.varfinemapysize;
            finemapxoffset = mapvarsin.varfinemapxoffset;
            finemapyoffset = mapvarsin.varfinemapyoffset;

            coarsemapxsize  = mapvarsin.varcoarsemapxsize;
            coarsemapysize = mapvarsin.varcoarsemapysize;
            coarsemapxoffset = mapvarsin.varcoarsemapxoffset;
            coarsemapyoffset = mapvarsin.varcoarsemapyoffset;
            coarsemapscale = mapvarsin.varcoarsemapscale;

            outdirectory = mapvarsin.outdirectory;

            lambda = mapvarsin.lambda;
            the_task = mapvarsin.the_task;
            desired_specnum = mapvarsin.desired_specnum;

            // pristine map information
            pristinefinemapinput = mapvarsin.pristinefinemapfile;
            pristinecoarsemapinput = mapvarsin.pristinecoarsemapfile;
            dPristine = mapvarsin.dPristine;
            dForestTransform = mapvarsin.dForestTransform;

            deme = mapvarsin.deme;
            deme_sample = mapvarsin.deme_sample;
            spec = mapvarsin.spec;
            zfat = mapvarsin.zfat;
            dLvalue = mapvarsin.dLvalue;
            maxtime = mapvarsin.maxtime;
            autocorrel_file = mapvarsin.autocorrel_file;
            varimport = true;
        }
        else
        {
            throw Main_Exception("ERROR_MAIN_001: Variables already imported.");
        }
        return;
    }

    vector<double> getTemporalSampling()
    {
        if(bAutocorrel)
        {
            return(autocorrel_times);
        }
        else
        {
            vector<double> tmp;
            tmp.push_back(0.0);
            return(tmp);
        }
    }

    // Imports the maps using the variables stored in the class. This function must␣
↪be run after the set_mapvars() in order to function correctly.
    void importMaps()
```

```
    {
        if(varimport)
        {
            // Set the dimensions
            try
            {
                forestmap.setDims(mapvarsin);
            }
            catch(Map_Fatal_Exception& mf)
            {
                cerr << mf.what() << endl;
                exit(-1);
            }
            try
            {
                // Set the time variables
                forestmap.checkMapExists();
                //forestmap.setTimeVars(dPristine,dForestTransform);
                // Import the fine map
                forestmap.calcFineMap();
                // Import the coarse map
                forestmap.calcCoarseMap();
                // Calculate the offset for the extremeties of each map
                forestmap.calcOffset();
                // Import the pristine maps;
                forestmap.calcPristineFineMap();
                forestmap.calcPristineCoarseMap();
            }
            catch(Map_Exception& me)
            {
                cerr << me.what() << endl;
                cerr << "No dimensions set - can't start simulations" << endl;
                exit(-1);
            }
            catch(Map_Fatal_Exception& mfe)
            {
                cerr << mfe.what() << endl;
                exit(-1);
            }

        }
        else
        {
            throw Fatal_Exception("ERROR_MAIN_002: Variables not imported.");
        }
    }

    long long getSeed()
    {
        return the_seed;
    }


    void setSeed(long long theseedin)
    {
        if (!seeded)
        {
            NR.setSeed(theseedin);
```

```
            the_seed = theseedin;
            seeded = true;
        }
    }

    vector<long> randomList(long maxnum , long numnum)
    {
        vector<long> isin;
        vector<long> isout;
        isin.clear();
        isout.clear();
        long endisout = maxnum + 1;
        for (long i = 0 ; i <= maxnum ; i ++)
        {
            isout.push_back(i);
        }
        while (unsigned(isin.size()) < numnum)
        {
            long chosen = NR.i0(endisout-1);
            isin.push_back(isout[chosen]);
            isout[chosen] = isout[endisout-1];
            endisout --;
        }

        return isin;
    }

    // New setup routine
    // This performs all set up jobs, importing the relevant files and creating the␣
↪in-memory objects for simulation start.
    void setup(vector<string>& comargs, bool fullmode)
    {
        cout << "***********************************************" << endl;
        cout << "Setting up simulation..." << endl;
        // deme = square root of Deme size - note deme*deme*number of demes is the␣
↪total size of the system.  Number of demes is given by the map scenario
        // deme_sample = number of individuals to be sampled from each deme (cannot␣
↪be more than deme^2)
        // spec = speciation rate required
        // dispersal = dispersal distance (double)
        // typeflag = 2 normal
        // typeflag != 2 fat
        // zfat = kernel fatness
        // map_scenario gives the index of the predefined map list vectors that will␣
↪indicate which habitat map to load
        // generations_since = number of generations since disturbance at present day␣
↪of sampling
        // equilibrium mask = do we ignore the generations_since tag and simply run␣
↪all the way to equilibrium on the fragmented landscape?
        // max time allowed for this simulation (useful for HPC runs)
        // Set the private variables for the simulation
        // Start the timer
        time(&start);
        // Use the Mapvars object to import the necessary information
         setMapVars(comargs, fullmode);

        try
        {
```

```
            if(autocorrel_file == "null")
            {
                bAutocorrel = false;
            }
            else
            {
                bAutocorrel = true;
                vector<string> tmpimport;
                ConfigOption tmpconfig;
                tmpconfig.setConfig(autocorrel_file,false);
                tmpconfig.importConfig(tmpimport);
                for(unsigned int i = 0;i<tmpimport.size();i++)
                {
                    autocorrel_times.push_back(stod(tmpimport[i]));
//                  cout << "t_i: " << autocorrel_times[i] << endl;
                }
            }
        }
        catch(Config_Exception& ce)
        {
            cerr << ce.what() << endl;
        }
        // Make the mask map with 1 entry in each deme
        // Previous versions worked with 1 entry per individual.
        try
        {
            importMaps(); // This will import the fine and coarse maps using the␣
↪routine specified in the Map class.
        }
        catch(Fatal_Exception& fe)
        {
            cerr << fe.what() << endl;
            throw Fatal_Exception();
        }
        // Set up the map recording information

        // Set the map dispersal values (so they don't have to be passed with every␣
↪function call).
        NR.setDispersalParams(zfat,dLvalue);
        if(zfat < 2.0)
        {
            cerr << "WARNING: zfat value is less than 2. This is likely to cause␣
↪severe problems with dispersal distances." << endl;
        }
        //forestmap.validateMaps();
        #ifdef debug_mode
        forestmap.validateMaps(); // If you're having problems with the maps␣
↪generating errors, run this line to check the maps are correctly formatted.
        #endif
        unsigned long initcount;
        // import the grid file
        samplegrid.importDatamask(mapvarsin);
        // Get a count of the number of individuals on the grid.
        try
        {
            initcount = forestmap.getInitialCount(deme_sample, samplegrid);
        }
        catch(exception& e)
```

```
        {
            cerr << e.what() << endl;
            throw Fatal_Exception();
        }
        // Set active and data at the correct sizes.
        if(initcount == 0)
        {
            cout << "Initial count is 0. No individuals to simulate. Exiting program.
↪" << endl;
            exit(0);
        }
        active.SetRowSize(initcount+1);
        data.SetRowSize(2*initcount+1);
        // Make the grid size with 1 entry per deme.
        // Previous versions used 1 entry per individual for increased spatial␣
↪movement. However, with percentage cover, this was now deemed unneccessary.
        grid.SetSize(gridysize,gridxsize);
        unsigned long numstart;
        numstart = 0;
        endactive =0;
        active[0].setup(0,0,0,0,0,0,0);
//      data[0].setSpec(1.0);
        cout << "\rSetting up simulation...filling grid                          " <
↪< flush;

        for (long i = 0 ; i < gridxsize ; i++)
        {
            for (long j = 0 ; j < gridysize ; j++)
            {
                grid[j][i].initialise(forestmap.getVal(i,j,0,0,0));
                grid[j][i].fillList();
                grid[j][i].setNwrap(0);
                grid[j][i].setNext(0);
                // loop over the species list, adding the correct number of species␣
↪in each placement.
                // Use the sampling to determine how many individuals to start in␣
↪each position.
                if(samplegrid.getVal(i,j))
                {
                    double dSample_amount = floor(deme_sample*grid[j][i].
↪getMaxsize());
                    for(int k=0;k<dSample_amount;k++)
                    {
                        if(k>=grid[j][i].getMaxsize())
                        {
                            break;
                        }
                        else
                        {
                            numstart ++;
                            int listpos = grid[j][i].addSpecies(numstart);
                            // Add the species to active
                            active[numstart].setup(i,j,0,0,numstart,listpos,1);
                            // Add a tip in the Treenode for calculation of the␣
↪coalescence tree at the end of the simulation.
                            // This also contains the start x and y position of the␣
↪species.
                            data[numstart].setup(1,i,j,0,0);
```

```
                            data[numstart].setSpec(NR.d01());
                            endactive ++;
                            enddata ++;
                        }
                    }
                }
            }
        }
//      cout << "check!" << endl;
        if(numstart == initcount) // Check that the two counting methods match up.
        {

        }
        else
        {
            if(initcount > 1.1*numstart)
            {
                cerr << "Data usage higher than neccessary - check allocation of␣
↪individuals to the grid." << endl;
            }
            // this error is silenced as of v3.1 due to a difference in the counting␣
↪methods of initcount and numstart when taking into account the sampling.
//          cout << "ERROR_MAIN_004: numstart not equal to initcount" << endl;
//          cout << "numstart: " << numstart << " initcount: " << initcount << endl;
//          cout << "endactive: " << endactive << " enddata: " << enddata << endl;
        }
        // other variables
        steps = 0;
        generation = 0;
        cout << "\rSetting up simulation...done!                                " << endl;
        cout << "Number of individuals simulated: " << endactive  << endl;
        maxsimsize = enddata;
//      cout << "Endactive: " << endactive << endl;
        if(active.size()<endactive || endactive == 0)
        {
            cerr << "endactive: " << endactive << endl;
            cerr << "active.size: " << active.size() << endl;
            cerr << "initcount: " << initcount << endl;
            cerr << "numstart: " <<numstart << endl;
            if(endactive ==0)
            {
                cout << "No individuals to simulate! Check map set up. Exiting..." <<␣
↪endl;
                exit(EXIT_SUCCESS);
            }
            else
            {
                throw Fatal_Exception("ERROR_MAIN_007: FATAL. Sizing error -␣
↪endactive is greater than the size of active.");
            }
        }
        #ifdef debug_mode
        // Checks that each lineage is placed in a legal place.
        for(unsigned int i=0;i<=endactive;i++)
        {
            if(forestmap.getVal(active[i].getXpos(),active[i].getYpos(),0,0,0)==0)
            {
                throw Fatal_Exception("ERROR_MAIN_006: FATAL.  Setup is placing␣
↪lineages in areas with no forest.");
```

```
            }
            if(active[i].getListpos()>grid[active[i].getYpos()][active[i].getXpos()].
↪getMaxsize())
            {
                cout << "FAIL2 IN " << i << endl;
                cout << "x,y: " << active[i].getXpos() << "," << active[i].getYpos() <
↪< " - listpos: " << active[i].getListpos() << " maxsize: " << grid[active[i].
↪getYpos()][active[i].getYpos()].getMaxsize() << endl;
                cout << "forestmap count: " << forestmap.getVal(active[i].getXpos(),
↪active[i].getYpos(),0,0,0) << endl;
                exit(0);
            }
        }

        // validate all lineages
        for(unsigned int i =1;i<=endactive;i++)
        {
            validationCheck(i,200,i);
        }
        #endif
    }


    void removeOldPos(const unsigned long &chosen)
    {
        long nwrap = active[chosen].getNwrap();
        long oldx = active[chosen].getXpos();
        long oldy = active[chosen].getYpos();
        if( nwrap== 0)
        {
            if(active[chosen].getXwrap() != 0 || active[chosen].getYwrap() != 0)
            {
                cerr << "chosen: " <<  chosen << endl;
                cerr << "x,y wrap: " << active[chosen].getXwrap() << "," <<␣
↪active[chosen].getYwrap() << endl;
                throw Fatal_Exception("ERROR_MOVE_015: Nwrap not set correctly. Nwrap␣
↪0, but x and y wrap not 0. " );
            }
            // Then the lineage exists in the main list;
            // debug (can be removed later)
            #ifdef pristine_mode
            if(grid[oldy][oldx].getMaxsize() < active[chosen].getListpos())
            {
                cerr << "grid maxsize: " << grid[oldy][oldx].getMaxsize() << endl;
                throw Fatal_Exception("ERROR_MOVE_001: Listpos outside maxsize. Check␣
↪move programming function.");
            }
            #endif
            // delete the species from the list
            grid[oldy][oldx].deleteSpecies(active[chosen].getListpos());
            // clear out the variables.
            active[chosen].setNext(0);
            active[chosen].setNwrap(0);
            active[chosen].setListpos(0);
        }
        else // need to loop over the nwrap to check nexts
        {
            if(nwrap==1)
```

```
            {
                grid[oldy][oldx].setNext(active[chosen].getNext());
                // Now reduce the nwrap of the lineages that have been effected.
                long nextpos = active[chosen].getNext();
                // loop over the rest of the list, reducing the nwrap
                while(nextpos!=0)
                {
                    active[nextpos].decreaseNwrap();
                    nextpos = active[nextpos].getNext();
                }
                // decrease the nwrap
                grid[oldy][oldx].decreaseNwrap();
                active[chosen].setNwrap(0);
                active[chosen].setNext(0);
                active[chosen].setListpos(0);
                nwrap =0;
            }
            else
            {
                long lastpos = grid[oldy][oldx].getNext();
                while(active[lastpos].getNext()!=chosen) // loop until we reach the␣
↪next, then set the next correctly.
                {
                    lastpos = active[lastpos].getNext();
                }
                if(lastpos!=0)
                {
                    active[lastpos].setNext(active[chosen].getNext());
                    // check
                    if(active[lastpos].getNwrap()!=(active[chosen].getNwrap()-1))
                    {
                        cerr << "lastpos : " << lastpos << " lastpos nwrap: " <<␣
↪active[lastpos].getNwrap() << endl;
                        cerr << "chosen: " << chosen << " chosen nwrap: " <<␣
↪active[chosen].getNwrap() << endl;
                        throw Fatal_Exception("ERROR_MOVE_022: nwrap setting of␣
↪either chosen or the lineage wrapped before chosen. Check move function.");
                    }
                    lastpos = active[lastpos].getNext();
                    while(lastpos!=0)
                    {
                        active[lastpos].decreaseNwrap();
                        lastpos = active[lastpos].getNext();
                    }
                }
                else
                {
                    cerr << "lastpos: " << lastpos << " endactive: " << endactive <<
↪" chosen: " << chosen << endl;
                    throw Fatal_Exception("ERROR_MOVE_024: Last position before␣
↪chosen is 0 - this is impossible.");
                }
                grid[oldy][oldx].decreaseNwrap();
                active[chosen].setNwrap(0);
                active[chosen].setNext(0);
                active[chosen].setListpos(0);
                nwrap =0;
            }
```

```
            unsigned short iCount =1;
            long pos = grid[oldy][oldx].getNext();
            if(pos==0)
            {
                iCount =0;
            }
            else
            {
                int c=0;
                while(active[pos].getNext()!=0)
                {
                    c++;
                    iCount ++;
                    pos = active[pos].getNext();
                    if(c>10000)
                    {
                        cout << pos << endl;
                        cout << active[pos].getNext() << endl;
                        break;
                    }
                }
            }

            if(iCount != grid[oldy][oldx].getNwrap())
            {
                cerr << "Nwrap: " << grid[oldy][oldx].getNwrap() << " Counted␣
↪lineages: " << iCount << endl;
                 throw Fatal_Exception("ERROR_MOVE_014: Nwrap not set correctly after␣
↪move for grid cell");
            }

        }
    }


    void calcMove(long &oldx, long &oldy, long &oldxwrap, long &oldywrap)
    {
        // initiate the move variables (distance and angle of movement)

        bool fail;
        fail = true;
        // keep looping until we reach a viable place to move from.
        while(fail)
        {
            #ifdef restrict_self
            long startx, starty, startxwrap, startywrap;
            startx = oldx;
            starty = oldy;
            startxwrap = oldxwrap;
            startywrap = oldywrap;
            #endif

            double dist, angle;
            #ifndef normal_dispersal
            dist = NR.fattail();


            #endif
```

```
            #ifdef normal_dispersal
            dist = zfat * NR.norm();
            #endif
//          if(dist > 1000)
//          {
//              cout << dist << endl;
//          }
//          long startx = oldx;
//          long starty = oldy;
//          angle = fmod(NR.direction(),2*M_PI);
            // Test different angle version - this is more correct (but for some␣
↪reason massively increases compute time).
            angle = NR.d01() * 2*M_PI;
            // check angle
//          if(angle < M_PI)
//          {
//              cout << "angle < pi" << endl;
//          }
//          else if(angle < 2*M_PI)
//          {
//              cout << "angle < 2*pi" <<endl;
//          }
//          else
//          {
//              cout << "angle > 2pi" << endl;
//          }
//          if(oldx <-1)
//          {
//              cout <<"x1: " << oldx << endl;
//          }
            forestmap.runDispersal(dist,angle,oldx,oldy,oldxwrap,oldywrap,fail,
↪generation);
//          if(oldx <-1)
//          {
//              cout <<"x1: " << oldx << endl;
//          }
//          if(dist > 1000)
//          {
//              cout << "start: " << startx << "," << starty << "pos: " << oldx << ",
↪" << oldy << " wrap: " << oldxwrap << "," << oldywrap << "fail: " << fail << endl;
//          }
            #ifdef restrict_self
            if(startx == oldx && starty == oldy && startxwrap == oldxwrap &&␣
↪startywrap == oldywrap)
            {
                fail = true;
            }
            #endif
            if(forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,generation)==0 && !fail)
            {
                cerr << "x,y: " << oldx << "," << oldy << " x,y wrap: " << oldxwrap <
↪< "," << oldywrap << endl << "Forest cover: " << forestmap.getVal(oldx,oldy,
↪oldxwrap,oldywrap,generation) << endl;
                throw Fatal_Exception("ERROR_MOVE_007: Dispersal attempted to non-
↪forest. Check dispersal function.");
            }
        }
    }
```

```
    //
    //
    long double calcMinMax(const unsigned long &current)
    {
        // this formula calculates the speciation rate required for speciation to␣
↪have occured on this branch.
        // need to allow for the case that the number of gens was 0
        long double newminmax =1;
        long double oldminmax = active[current].getMinmax();
        if(data[active[current].getMpos()].getGenRate()==0)
        {
            newminmax = data[active[current].getMpos()].getSpecRate();
        }
        else
        {
            // variables need to be defined separately for the decimal division to␣
↪function properly.
            long double tmpdSpec = data[active[current].getMpos()].getSpecRate();
            long double tmpiGen = data[active[current].getMpos()].getGenRate();
            newminmax = 1-(pow(1-tmpdSpec,(1/tmpiGen)));
        }
        long double toret = min(newminmax,oldminmax);
        return toret;
    }

    void coalescenceEvent(const unsigned long &chosen, unsigned long &coalchosen)
    {
        // coalescence occured, so we need to adjust the data appropriatedly
        // our chosen lineage has merged with the coalchosen lineage, so we need to␣
↪sync up the data.
        enddata ++;
        data[enddata].setup(0,active[chosen].getXpos(),active[chosen].getYpos(),
↪active[chosen].getXwrap(),active[chosen].getYwrap());

        // First perform the move
        data[active[chosen].getMpos()].setParent(enddata);
        data[active[coalchosen].getMpos()].setParent(enddata);
        active[coalchosen].setMinmax(max(active[coalchosen].getMinmax(),
↪active[chosen].getMinmax())); // set the new minmax to the maximum of the two␣
↪minimums.
        active[chosen].setMinmax(active[coalchosen].getMinmax());
        data[enddata].setIGen(0);
        data[enddata].setSpec(NR.d01());
        active[chosen].setMpos(enddata);
        active[coalchosen].setMpos(enddata);
//      removeOldPos(chosen);
        switchPositions(chosen);
    }


    //
    //
    void calcNewPos(bool &coal, const unsigned long &chosen,unsigned long &coalchosen,
↪ const long &oldx, const long &oldy, const long &oldxwrap, const long &oldywrap)
    {
        // Calculate the new position of the move, whilst also calculating the␣
↪probability of coalescence.
```

```
        unsigned short nwrap = active[chosen].getNwrap();
        if(oldxwrap == 0 && oldywrap == 0)
        {
            // Debug check (to remove later)
            if(nwrap!=0)
            {
                throw Fatal_Exception("ERROR_MOVE_006: NON FATAL. Nwrap not set␣
↪correctly. Check move programming function.");
            }
            // then the procedure is relatively simple.
            // check for coalescence
            // check if the grid needs to be updated.
            if(grid[oldy][oldx].getMaxsize()!=forestmap.getVal(oldx,oldy,oldxwrap,
↪oldywrap,generation))
            {
                grid[oldy][oldx].setMaxsize(forestmap.getVal(oldx,oldy,0,0,
↪generation));
            }
            coalchosen = grid[oldy][oldx].getRandLineage(NR);
            //#ifdef DEBUG
            if(coalchosen !=0)
            {
                if(active[coalchosen].getXpos() != (unsigned long) oldx ||␣
↪active[coalchosen].getYpos() != (unsigned long) oldy || active[coalchosen].
↪getXwrap() != oldxwrap || active[coalchosen].getYwrap() != oldywrap)
                {
                    cerr << chosen << "," << coalchosen << endl;
                    cerr << "chosen - x,y: " << oldx << "," << oldy << endl << "x, y␣
↪wrap: " << oldxwrap << "," << oldywrap << endl;
                    cerr << "coalchosen - x,y:" << active[coalchosen].getXpos() << ",
↪" << active[coalchosen].getYpos() << endl;
                    cerr <<" x,y wrap: " << active[coalchosen].getXwrap() << "," <<␣
↪active[coalchosen].getYwrap() << endl;
                    throw Fatal_Exception("ERROR_MOVE_006: NON FATAL. Nwrap not set␣
↪correctly. Check move programming function.");
                }
            }
            //#endif
            if(coalchosen == 0) // then the lineage can be placed in the empty space.
            {
                long tmplistindex = grid[oldy][oldx].addSpecies(chosen);
                //check
                if(grid[oldy][oldx].getSpecies(tmplistindex)!=chosen)
                {
                    throw Fatal_Exception("ERROR_MOVE_005: Grid index not set␣
↪correctly for species. Check move programming function.");
                }
                #ifdef pristine_mode
                if(grid[oldy][oldx].getListsize()> grid[oldy][oldx].getMaxsize())
                {
                    throw Fatal_Exception("ERROR_MOVE_001: Listpos outside maxsize.␣
↪Check move programming function.");
                }
                #endif
                active[chosen].setNwrap(0);
                active[chosen].setListpos(tmplistindex);
                coal=false;
            }
```

```
            else // then coalescence has occured
            {
                active[chosen].setNwrap(0);
                active[chosen].setListpos(0);
                // DO THE COALESCENCE STUFF
                coal = true;
            }
        }
        else // need to check all the possible places the lineage could be.
        {
            if(nwrap!=0)
            {
                throw Fatal_Exception("ERROR_MOVE_022: Nwrap not set correctly in␣
↪move.");
            }
            nwrap = grid[oldy][oldx].getNwrap();
            if(nwrap != 0) // then coalescence is possible and we need to loop over␣
↪the nexts to check those that are in the same position
            {
                // Count the possible matches of the position.
                int matches=0;
                // Create an array containing the list of active references for those␣
↪that match as
                // this stops us having to loop twice over the same list.
                unsigned long matchlist [nwrap];
                long next_active;
                next_active = grid[oldy][oldx].getNext();
                // Count if the first "next" matches
                if(active[next_active].getXwrap()==oldxwrap && active[next_active].
↪getYwrap()==oldywrap)
                {
                    // check
                    #ifdef DEBUG
                    if(active[next_active].getNwrap() != 1)
                    {
                        throw Fatal_Exception("ERROR_MOVE_022a: Nwrap not set␣
↪correctly in move.");
                    }
                    #endif
                    matchlist[matches] = next_active; // add the match to the list of␣
↪matches.
                    matches ++;
                }
                // Now loop over the remaining nexts counting matches
                //#ifdef DEBUG
                unsigned int ncount =1;
                //#endif
                while(active[next_active].getNext()!=0)
                {
                    next_active = active[next_active].getNext();
                    if(active[next_active].getXwrap()==oldxwrap && active[next_
↪active].getYwrap()==oldywrap)
                    {
                        matchlist[matches] = next_active;
                        matches ++;
                    }
                    // check
                    //#ifdef DEBUG
```

```
                    ncount ++;
                    #ifdef DEBUG
                    if(active[next_active].getNwrap() != ncount)
                    {
                        throw Fatal_Exception("ERROR_MOVE_022d: Nwrap not set␣
↪correctly in move.");
                    }
                    #endif
                }
                if(nwrap != ncount)
                {
                    throw Fatal_Exception("ERROR_MOVE_022c: Nwrap not set correctly␣
↪in move.");
                }
                // Matches now contains the number of lineages at the exact x,y,␣
↪xwrap and ywrap position.
                // Check if there were no matches at all
                if(matches==0)
                {
                    coalchosen = 0;
                    coal = false;
                    active[next_active].setNext(chosen);
                    grid[oldy][oldx].increaseNwrap();
                    active[chosen].setNwrap(grid[oldy][oldx].getNwrap());
                    active[chosen].setListpos(0);
                }
                else // if there were matches, generate a random number to see if␣
↪coalescence occured or not
                {
                    unsigned short randwrap = floor(NR.d01()*(forestmap.getVal(oldx,
↪oldy,oldxwrap,oldywrap,generation))+1);
                    // Get the random reference from the match list.
                    // If the movement is to an empty space, then we can update the␣
↪chain to include the new lineage.
                    if(randwrap > forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,
↪generation))
                    {
                        throw Fatal_Exception("ERROR_MOVE_004: Randpos outside␣
↪maxsize. Check move programming function");
                    }
                    if(matches > forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,
↪generation))
                    {
                        cerr << "matches: " << matches << endl << "forestmap value: "
↪<< forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,generation);
                        throw Fatal_Exception("ERROR_MOVE_004: matches outside␣
↪maxsize. Check move programming function");
                    }
                    if(randwrap > matches) // coalescence has not occured
                    {
                        //cout << "This shouldn't happen" << endl;
                        coalchosen = 0;
                        coal = false;
                        active[next_active].setNext(chosen);
                        grid[oldy][oldx].increaseNwrap();
                        active[chosen].setNwrap(grid[oldy][oldx].getNwrap());
                        active[chosen].setListpos(0);
                    }
```

```
                    else // coalescence has occured
                    {
                        coal = true;
                        coalchosen = matchlist[randwrap-1];
                        active[chosen].setEndpoint(oldx,oldy,oldxwrap,oldywrap);
                        if(coalchosen==0)
                        {
                            throw Fatal_Exception("ERROR_MOVE_025: Coalescence␣
↪attempted with lineage of 0.");
                        }
                    }
                }
                #ifdef pristine_mode
                if(grid[oldy][oldx].getMaxsize()<active[chosen].getListpos())
                {
                    throw Fatal_Exception("ERROR_MOVE_001: Listpos outside maxsize.␣
↪Check move programming function.");
                }
                #endif
            }
            else // just add the lineage to next.
            {
                if(grid[oldy][oldx].getNext()!=0)
                {
                    throw Fatal_Exception("ERROR_MOVE_026: No nwrap recorded, but␣
↪next is non-zero.");
                }
                coalchosen = 0;
                coal = false;
//              if(chosen==3893119)
//              {
//                  validationCheck(31556348,12,chosen);
//                  cout << "test1..." << endl;
//              }
//              debug_504(chosen,12);
                grid[oldy][oldx].setNext(chosen);
                active[chosen].setNwrap(1);
                active[chosen].setNext(0);
                grid[oldy][oldx].increaseNwrap();
                // check
                #ifdef DEBUG
                 if(grid[oldy][oldx].getNwrap() != 1)
                 {
                     throw Fatal_Exception("ERROR_MOVE_022b: Nwrap not set correctly␣
↪in move.");
                 }
                 #endif
//              debug_504(chosen,11);
//              if(chosen==3893119)
//              {
//                  validationCheck(31556348,13,chosen);
//                  cout << "test2..." << endl;
//              }
            }
            //#ifdef DEBUG
            if(coalchosen !=0)
            {
                if(active[coalchosen].getXpos() != (unsigned long) oldx ||␣
↪active[coalchosen].getYpos() != (unsigned long) oldy || active[coalchosen].
↪getXwrap() != oldxwrap || active[coalchosen].getYwrap() != oldywrap)
```

```
                {
                    cerr << chosen << "," << coalchosen << endl;
                    cerr << "chosen - x,y: " << oldx << "," << oldy << endl << "x, y␣
↪wrap: " << oldxwrap << "," << oldywrap << endl;
                    cerr << "coalchosen - x,y:" << active[coalchosen].getXpos() << ",
↪" << active[coalchosen].getYpos() << endl;
                    cerr <<" x,y wrap: " << active[coalchosen].getXwrap() << "," <<␣
↪active[coalchosen].getYwrap() << endl;
                    throw Fatal_Exception("ERROR_MOVE_006b: NON FATAL. Nwrap not set␣
↪correctly. Check move programming function.");
                }
            }
            //#endif
        }
    }
    // This function is useless now (as it was only useful for debugging purposes)␣
↪and will be removed in later versions.
    /*
    {
        long nwrap = grid[504][0].getNwrap();
        unsigned long gridnext = grid[504][0].getNext();
        unsigned short tmpnwrap =0;
        while(gridnext!=0)
        {
            tmpnwrap++;
            gridnext = active[gridnext].getNext();
        }
        if(tmpnwrap!=nwrap)
        {
            cout << "TMP: " << endl;
            cout << "nwrap: " << nwrap << endl;
            cout << "tmpnwrap: " << tmpnwrap << endl;
            cout << "next: " << gridnext << endl;
            cout << "c: " << c << endl;
            string sam;
            cin >> sam;
        }
    }
     //*/
    //
    void switchPositions(const unsigned long chosen)
    {
        if(chosen>endactive)
        {
            cerr << "chosen: " << chosen << " endactive: " << endactive << endl;
            throw Fatal_Exception("ERROR_MOVE_023: Chosen is greater than endactive.␣
↪Check move function.");
        }
        if(chosen!=endactive)
        {
            // This routine assumes that the previous chosen position has already␣
↪been deleted.
            Datapoint tmpdatactive;
            tmpdatactive.setup(active[chosen]);
            // now need to remove the chosen lineage from memory, by replacing it␣
↪with the lineage that lies in the last place.
            if(active[endactive].getXwrap() ==0&&active[endactive].getYwrap()==0) //␣
↪if the end lineage is simple, we can just copy it across.
```

```
            {
                // check endactive
                if(active[endactive].getNwrap()!=0)
                {
                    cerr << "ERROR_MOVE_020: NON FATAL. Nwrap is not set correctly␣
↪for endactive (nwrap should be 0, but is " << active[endactive].getNwrap() << " ).␣
↪Identified during switch of positions." << endl;
                }
                grid[active[endactive].getYpos()][active[endactive].getXpos()].
↪setSpecies(active[endactive].getListpos(),chosen);
                active[chosen].setup(active[endactive]);
                active[endactive].setup(tmpdatactive);
                active[endactive].setNwrap(0);
                active[endactive].setNext(0);

            }
            else // else the end lineage is wrapped, and needs to be processed␣
↪including the wrapping routines.
            {
                if(active[endactive].getNwrap()==0)
                {
                    cerr << "ERROR_MOVE_021: NON FATAL. Nwrap is not set correctly␣
↪for endactive (nwrap incorrectly 0). Identified during switch of positions." <<␣
↪endl;
                }
//              cout << "wrap"<<endl;
                long tmpactive = grid[active[endactive].getYpos()][active[endactive].
↪getXpos()].getNext();
                int  tmpnwrap = active[endactive].getNwrap();

                // if the wrapping is just once, we need to set the grid next to the␣
↪chosen variable.
                if(tmpnwrap==1)
                {
                    // check
                    if(grid[active[endactive].getYpos()][active[endactive].getXpos()].
↪getNext()!=endactive)
                    {
                        throw Fatal_Exception(string("ERROR_MOVE_019: FATAL. Nwrap␣
↪for endactive not set correctly. Nwrap is 1, but lineage at 1st position is " +
                        to_string((long long)grid[active[endactive].
↪getYpos()][active[endactive].getXpos()].getNext()) + ". Identified during the move.
↪"));
                    }
                    grid[active[endactive].getYpos()][active[endactive].getXpos()].
↪setNext(chosen);
                }
                else // otherwise, we just set the next to chosen instead of␣
↪endactive.
                {
                    int tmpcount = 0;
                    // loop over nexts until we reach the right lineage.
                    while(active[tmpactive].getNext()!=endactive)
                    {
                        tmpactive = active[tmpactive].getNext();
                        tmpcount ++;
                        // debug check
                        if(tmpcount > tmpnwrap)
```

```
                        {
                            cerr << "ERROR_MOVE_013: NON FATAL. Looping has not␣
↪encountered a match, despite going further than required. Check nwrap counting." <<␣
↪endl;
                            if(tmpactive==0)
                            {
                                cerr << "gridnext: " << grid[active[endactive].
↪getYpos()][active[endactive].getXpos()].getNext() << endl;
                                cerr << "x,y: " << active[endactive].getXpos() << ","
↪<< active[endactive].getYpos() << endl;
                                cerr << "xwrap,ywrap: " << active[endactive].
↪getXwrap() << "," << active[endactive].getYwrap() << endl;
                                cerr << "endactive: " << endactive << endl;
                                cerr << "tmpactive: " << tmpactive << endl;
                                cerr << "bSpec: " << bSpec << endl;
                                cerr << "tmpnwrap: " <<  tmpnwrap << " tmpcount: " <<␣
↪tmpcount << endl;
                                cerr << "FATAL!" << endl;
                                throw Fatal_Exception();
                            }
                        }

                    }
                    active[tmpactive].setNext(chosen);
                }
                active[chosen].setup(active[endactive]);
                active[endactive].setup(tmpdatactive);

                // check - debugging
                long testwrap = active[chosen].getNwrap();
                unsigned long testnext = grid[active[chosen].
↪getYpos()][active[chosen].getXpos()].getNext();
                for(int i=1;i<testwrap;i++)
                {
                    testnext = active[testnext].getNext();
                }

                if(testnext!=chosen)
                {
                    throw Fatal_Exception("ERROR_MOVE_009: Nwrap position not set␣
↪correctly after coalescence. Check move process.");
                }
            }
        }
        endactive --;
    }
    void speciation(const unsigned long &chosen)
    {
        // alter the data such that it reflects the speciation event.
        unsigned long tmpmpos = active[chosen].getMpos();
        //data[tmpmpos].increaseGen();
        #ifdef DEBUG
        if(data[tmpmpos].hasSpeciated())
        {
            throw Fatal_Exception("ERROR_MOVE_028: Attempting to speciate a speciated␣
↪species.");
        }
        #endif
```

```
        data[tmpmpos].speciate();
        // TEST REMOVE THIS WHEN TESTING COMPLETE!! done
//      data[tmpmpos].setPosition(active[chosen].getXpos(),active[chosen].getYpos(),
↪active[chosen].getXwrap(),active[chosen].getYwrap());
        // Now remove the old chosen lineage from the active directory.
        removeOldPos(chosen);
        switchPositions(chosen);
    }


    unsigned long estSpecnum()
    {
        long double dMinmax = 0;
        // first loop to find the maximum speciation rate required
        for(unsigned int i =1;i<=endactive;i++)
        {
            long double tmpminmax = calcMinMax(i);
            active[i].setMinmax(tmpminmax);
            dMinmax = (long double)max(dMinmax,tmpminmax);
        }
        for(unsigned int i=0;i<=enddata;i++)
        {
            if(data[i].isTip())
            {
                data[i].setExistance(true);
            }
            double maxret = 1;
            if(data[i].getGenRate()==0)
            {
                maxret = 1;
            }
            else
            {
                maxret = data[i].getGenRate();
            }
            // This is the line that compares the individual random numbers against␣
↪the speciation rate.
            if(data[i].getSpecRate()<(1-pow((1-dMinmax),maxret)))
            {
                data[i].speciate();
            }
        }
        bool loop=true;
        while(loop)
        {
            loop=false;
            for(unsigned int i=0;i<=enddata;i++)
            {
                if(data[i].getExistance()&&!data[data[i].getParent()].getExistance()&&
↪!data[i].hasSpeciated())
                {
                    loop = true;
                    data[data[i].getParent()].setExistance(true);
                }
            }
        }
        unsigned long iSpecies=0;
        for(unsigned int i=0;i<=enddata;i++)
```

```
        {
            if(data[i].getExistance()&&data[i].hasSpeciated())
            {
                iSpecies ++;
            }
        }
        for(unsigned int i=0;i<=enddata;i++)
        {
            data[i].qReset();
        }
//      cout << "Estimated species number is: " << iSpecies << endl;
        return iSpecies;
    }


    void runChecks(const unsigned long &chosen, const unsigned long &coalchosen)
    {
        // final checks
        #ifdef pristine_mode
        if(active[chosen].getListpos() > grid[active[chosen].
↪getYpos()][active[chosen].getXpos()].getMaxsize()&&active[chosen].getNwrap()==0)
        {
//              usleep(1);
            cerr << "listpos: " << active[chosen].getListpos() << " maxsize: " <<␣
↪grid[active[chosen].getYpos()][active[chosen].getXpos()].getMaxsize() <<endl;
            throw Fatal_Exception("ERROR_MOVE_001: Listpos outside maxsize.");
        }

        if(active[coalchosen].getListpos() > grid[active[coalchosen].
↪getYpos()][active[coalchosen].getXpos()].getMaxsize()&&active[coalchosen].
↪getNwrap()==0&&coalchosen!=0)
        {
//              usleep(1);
            throw Fatal_Exception("ERROR_MOVE_002: Coalchosen listpos outside maxsize.
↪");
        }
        #endif
        if(active[chosen].getNwrap()!=0)
        {
            unsigned long tmpactive = grid[active[chosen].getYpos()][active[chosen].
↪getXpos()].getNext();
            unsigned long lastactive = 0;
            for(int i=1;i<active[chosen].getNwrap();i++)
            {
                lastactive = tmpactive;
                tmpactive = active[tmpactive].getNext();
            }
            if(tmpactive!=chosen)
            {
                cerr << "nwrap: " << active[chosen].getNwrap() << endl;
                cerr << "gridnwrap: " << grid[active[chosen].
↪getYpos()][active[chosen].getXpos()].getNwrap() << endl;
                cerr << "chosen: " << chosen << endl;
                cerr << "active x,y" << active[chosen].getXpos() << "," <<␣
↪active[chosen].getYpos() << endl;
                cerr << "wrap x,y: " << active[chosen].getXwrap() << "," <<␣
↪active[chosen].getYwrap() << endl;
                cerr << "tmpactive: " << tmpactive << " chosen: " << chosen << "␣
↪chosennext: " << active[chosen].getNext() << "lastactive: " << lastactive<< endl;
```

```
                throw Fatal_Exception("ERROR_MOVE_003: Nwrap not set correctly.");
            }
        }

        if(active[chosen].getNwrap()!=0)
        {
            if(active[chosen].getXwrap()==0&&active[chosen].getYwrap()==0)
            {
                throw Fatal_Exception("ERROR_MOVE_10: Nwrap set to non-zero, but x␣
↪and y wrap 0.");
            }
        }
        if(active[endactive].getNwrap()!=0)
        {
            int nwrap = active[endactive].getNwrap();
            if(nwrap==1)
            {
                if(grid[active[endactive].getYpos()][active[endactive].getXpos()].
↪getNext()!=endactive)
                {
                    cerr << "Lineage at 1st position: " << grid[active[endactive].
↪getYpos()][active[endactive].getXpos()].getNext() << endl;
                    cerr << "endactive: " << endactive << endl <<"nwrap: " << nwrap <
↪< endl << "x,y: " << active[endactive].getXpos() << "," << active[endactive].
↪getYpos() << endl;
                    cerr << "chosen: " << chosen << endl;
                    throw Fatal_Exception("ERROR_MOVE_016: Nwrap for endactive not␣
↪set correctly. Nwrap is 1, but the lineage at 1st position is not endactive.");
                }
            }
            else
            {
                unsigned long tmpcheck = grid[active[endactive].
↪getYpos()][active[endactive].getXpos()].getNext();
                int tmpnwrap = 1;
                while(tmpcheck!=endactive)
                {
                    tmpnwrap ++;
                    tmpcheck = active[tmpcheck].getNext();
                    if(tmpnwrap>nwrap+1)
                    {
                        cerr << "ERROR_MOVE_017: NON FATAL. Nrap for endactive not␣
↪set correctly; looped beyond nwrap and not yet found enactive." << endl;
                        cerr << "endactive: " << endactive << endl <<"nwrap: " <<␣
↪nwrap << endl << "x,y: " << active[endactive].getXpos() << "," << active[endactive].
↪getYpos() << endl;
                        cerr << "chosen: " << chosen << endl;
                    }
                }
                if(tmpnwrap!=nwrap)
                {
                    cerr << "ERROR_MOVE_018: NON FATAL. Nwrap for endactive not set␣
↪correctly. Nwrap is " << nwrap << " but endactive is at position " << tmpnwrap <<␣
↪endl;
                    cerr << "endactive: " << endactive << endl <<"nwrap: " << nwrap <
↪< endl << "x,y: " << active[endactive].getXpos() << "," << active[endactive].
↪getYpos() << endl;
                    cerr << "chosen: " << chosen << endl;
```

```
                }

            }
        }
    }

    void validationCheck(const unsigned long &chosen,int o, const unsigned long &
↪current, const bool &coal)
    {
        //if(active[chosen].getNwrap()!=0||chosen>endactive)
        //{
        //  return;
        //}
//      cout << "check..." << endl;
        if(active[chosen].getListpos() > grid[active[chosen].
↪getYpos()][active[chosen].getXpos()].getMaxsize()&&active[chosen].getNwrap()==0)
        {
//              usleep(1);
            cout << "listpos: " << active[chosen].getListpos() << " maxsize: " <<␣
↪grid[active[chosen].getYpos()][active[chosen].getXpos()].getMaxsize() <<endl;
            cout << "VALIDATION_001: Listpos outside maxsize." << endl;
            throw Fatal_Exception("VALIDATION_001: Listpos outside maxsize.");
        }
        if(!coal && !data[active[chosen].getMpos()].hasSpeciated() && active[chosen].
↪getNwrap() ==0 && chosen!=grid[active[chosen].getYpos()][active[chosen].getXpos()].
↪getSpecies(active[chosen].getListpos()))
        {
            cout << "nwrap: " << active[chosen].getNwrap() << endl;
            cout << "gridnwrap: " << grid[active[chosen].getYpos()][active[chosen].
↪getXpos()].getNwrap() << endl;
            cout << "chosen: " << chosen << endl;
            cout << "active x,y: " << active[chosen].getXpos() << "," <<␣
↪active[chosen].getYpos() << endl;
            cout << "wrap x,y: " << active[chosen].getXwrap() << "," <<␣
↪active[chosen].getYwrap() << endl;
            cout << " chosen: " << chosen << " chosennext: " << active[chosen].
↪getNext() << endl;
             cout << "listpos: " << active[chosen].getListpos() << endl;
             cout << "lineage at pos " << active[chosen].getListpos() << ": " <<␣
↪grid[active[chosen].getYpos()][active[chosen].getXpos()].getSpecies(active[chosen].
↪getListpos()) << endl;
            cout << "o: " << o << endl;
            cout << "current chosen: " << current << endl;
            throw Fatal_Exception("VALIDATION_003: Listpos not set correctly.");
        }
        if(active[chosen].getNwrap()!=0)
        {
//              cout << "nwrap1: " << active[chosen].getNwrap() << endl;
            unsigned long tmpactive = grid[active[chosen].getYpos()][active[chosen].
↪getXpos()].getNext();
            unsigned long lastactive = 0;
            for(int i=1;i<active[chosen].getNwrap();i++)
            {
                lastactive = tmpactive;
                tmpactive = active[tmpactive].getNext();
            }
            if(tmpactive!=chosen)
            {
```

```
                cout << "nwrap: " << active[chosen].getNwrap() << endl;
                cout << "gridnwrap: " << grid[active[chosen].
↪getYpos()][active[chosen].getXpos()].getNwrap() << endl;
```

```
//              cout << "speccounter: " << spec_counter << endl;
//              cout << "startnwrap: " << startnwrap << endl;
                cout << "chosen: " << chosen << endl;
//              cout << "start x,y: " << startx << "," << starty << endl;
//              cout << "end x,y: " << oldx << "," << oldy << endl;
                cout << "active x,y: " << active[chosen].getXpos() << "," <<␣
↪active[chosen].getYpos() << endl;
                cout << "wrap x,y: " << active[chosen].getXwrap() << "," <<␣
↪active[chosen].getYwrap() << endl;
                cout << "tmpactive: " << tmpactive << " chosen: " << chosen << "␣
↪chosennext: " << active[chosen].getNext() << "lastactive: " << lastactive<< endl;
                cout << "VALIDATION_002: Nwrap not set correctly." << endl;
                cout << "o: " << o << endl;
                cout << "current chosen: " << current << endl;
                throw Fatal_Exception("VALIDATION_001: Listpos outside maxsize.");
            }
        }
    }


    void checkSimSize(unsigned long req_data,unsigned long req_active)
    {
//      cout << "Started change size" << endl;
        // need to be double the size of the maximum number of individuals plus␣
↪enddata
        unsigned long min_data = (2*req_data) + enddata;
        unsigned long min_active = endactive + req_active;
        if(data.size() < min_data)
        {
            // change the size of data
            data.changeSize(min_data);
        }

        if(active.size() < min_active)
        {
            // change the size of active.
            active.changeSize(min_active);
        }
//      cout << "finished change size" << endl;
    }

    bool runSim()
    {
        // now do the calculations required to build the tree
        cout << "************************************************" << endl;
        cout << "Beginning simulations..." << flush;
//      double current_gen =0;
        startendactive = endactive;
        // check time
        time(&sim_start);
        time(&sim_end);
        time(&now);
        time(&tCheckSpec);
        bool bContinueSim=true;
```

```
        #ifdef verbose
        double dPercentComplete;
        #endif
        unsigned int iAutoComplete = 0;
//      time_t loopend;
        // Main while loop to process while there is still time left and the simulation is not complete.
        while ((endactive > 1)&&(difftime(sim_end,start) < maxtime)&&bContinueSim)
        {
            // check the size of the simulations - decided this doesn't need to be here if the check is performed properly at map expansion.
//          checkSimSize(100,false,0);
//          time(&loopstart);
            steps ++;
            #ifdef verbose
            if ((steps%10000==0)||debug)
            {
                time(&sim_end);
                if(sim_end-now > 0.2&&log_all) // output every 0.2 seconds
                {

                    dPercentComplete = 100*(1-(double(endactive)/double(startendactive)));
                    time(&now);

                    cout  << "\rBeginning simulations..."<<endactive << " lineages remaining: " << dPercentComplete << "%
"       " << flush;

                }
            }
            #endif
            // increment generation counter
            generation += 2.0/(double(endactive));
            if(bAutocorrel && iAutoComplete < autocorrel_times.size())
            {
                //cout << "Check1" << endl;
//              cout << iAutoComplete << endl;
//              cout << autocorrel_file << endl;
//              for (int i = 0;i<autocorrel_times.size(); i ++)
//              {
//                  cout << autocorrel_times[i] << endl;
//              }
                // check if we need to update
                if(autocorrel_times[iAutoComplete]<=generation)
                {
//                  cout << "check2" << endl;
                    if(autocorrel_times[iAutoComplete]>0.0)
                    {
                        cout << "expanding map at generation " << generation << "..." << flush;
                        expandMap(autocorrel_times[iAutoComplete]);
                        cout << "done!" << endl;
                    }
                    iAutoComplete ++;
                }
            }
            // check if the map is pristine yet
```

```
            if(generation>=dPristine)
            {
                forestmap.setPristine(true);
            }
            // choose a random lineage to die and be reborn out of those currently␣
↪active
            unsigned long chosen = NR.i0(endactive-1)+1; // cannot be 0
            unsigned long coalchosen = 0;
            // record old position of lineage
            long oldx = active[chosen].getXpos();
            long oldy = active[chosen].getYpos();
//          long startxpos, startypos;
//          startxpos = oldx;
//          startypos = oldy;
            long oldxwrap = active[chosen].getXwrap();
            long oldywrap = active[chosen].getYwrap();
            bool coal = false;
            #ifdef pristine_mode
            if(forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,generation)==0)
            {
                cerr << "x,y: " << oldx << "," << oldy << " xwrap, ywrap: " <<␣
↪oldxwrap << "," << oldywrap << endl;
                cerr << "listsize: " << grid[oldy][oldx].getListsize() << "maxsize: "
↪<< grid[oldy][oldx].getMaxsize() << endl;
                throw Fatal_Exception(string("ERROR_MOVE_008: Dispersal attempted␣
↪from non-forest. Check dispersal function. Forest cover: " + to_string((long␣
↪long)forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,generation))));
            }
            #endif
            // This bit has been removed as it has a very significant performance hit␣
↪and is not required for most simulations. As of version 3.2 it was fully compatible␣
↪with the rest of the simulation, however.
            /*
            if(steps%1000000==0)
            {
                time(&now);
                if(now - tCheckSpec>200&&dPercentComplete>95)
                {
                    time(&tCheckSpec);
                    unsigned long specnum = est_specnum();
                    cout << "Estimated number of species: " << specnum << flush;
                    if(specnum<desired_specnum)
                    {
                        cout << " - desired number of species reached." << endl <<
↪"Halting simulations..." << endl;
                        bContinueSim = false;
                    }
                    else
                    {
                        cout << endl;
                    }
                }
            }
            //*/
            // LIST OF POTENTIAL IMPROVEMENTS FOR THE FUTURE
            // 1 - Improve the SpeciesList referencing method when a species is␣
↪deleted from the list.
            // 2 - Improve how the looping over next and last occurs so that the loop␣
↪only needs to be performed once. This could have major speed implications.
```

```
            // decide if speciation occured
            // Include alteration of forestmap over time.
            // Continue the simulation if the conditions are met for bContinueSim (in␣
↪test scenarios this will always be true until all lineages have coalesced.
            #ifdef DEBUG
            string location = "none";
            #endif
            try
            {
                if(bContinueSim)
                {
                    //cout << "check1" << endl;
                    coal = false;
                    // increase the counter of the number of moves (or generations)␣
↪the lineage has undergone.
                    data[active[chosen].getMpos()].increaseGen();
                    // Check if speciation happens
                    if(checkSpeciation(data[active[chosen].getMpos()].getSpecRate(),␣
↪spec, data[active[chosen].getMpos()].getGenRate()))
                    {
                        //cout << "spec: " << spec << endl;
                        //cout << "data spec rate: " << data[active[chosen].
↪getMpos()].getSpecRate() << endl;
                        //cout << "iGen: " << data[active[chosen].getMpos()].
↪getGenRate() << endl;
                        //string sam;
                        //cin >> sam;
                        #ifdef DEBUG
                        location = "speciation";
                        //if ( data[active[chosen].getMpos()].)
                        #endif
                        speciation(chosen);
                    }
                    else
                    {
                        #ifdef DEBUG
                        location = "standard: oldpos " + to_string(oldx) + to_
↪string(oldy);
                        #endif
                        removeOldPos(chosen); // remove the species data from the␣
↪species list to be placed somewhere new.
//                      long oldoldxwrap = oldxwrap;
                                        //cout << "check1.b" << endl;

                        calcMove(oldx,oldy,oldxwrap,oldywrap);
//                      if(oldoldxwrap != oldxwrap)
//                      {
//                          cout << "CHANGED!" << oldoldxwrap << "," << oldxwrap <<␣
↪endl;
//                      }
                                        //cout << "check2.a" << endl;

                        // Calculate the new position, perform the move if␣
↪coalescence doesn't occur or return the variables for the coalescence event if␣
↪coalescence does occur.
                        active[chosen].setEndpoint(oldx,oldy,oldxwrap,oldywrap); //␣
↪the "old" variables have been updated, so we can just input them back in to the␣
↪function.
```

```
                        calcNewPos(coal,chosen,coalchosen,oldx,oldy,oldxwrap,
↪oldywrap);
                        //cout << "check2.b" << endl;
                        if(coal) // coalescence occured, so we need to adjust the␣
↪data appropriatedly
                        {
                            #ifdef DEBUG
                            location = "coalescence: coalchosen - " + to_
↪string(coalchosen);
                            if(active[coalchosen].getXpos() != active[chosen].
↪getXpos() || active[coalchosen].getYpos() != active[chosen].getYpos()
                                || active[coalchosen].getXwrap() != active[chosen].
↪getXwrap() || active[coalchosen].getYwrap() != active[chosen].getYwrap())
                            {
                                cerr << chosen << "," << coalchosen << endl;
                                cerr << "chosen - x,y: " << active[chosen].getXpos() <
↪< "," << active[chosen].getYpos() << endl << "x, y wrap: " << active[chosen].
↪getXwrap();
                                cerr << "," << active[chosen].getYwrap() << endl;
                                cerr << "coalchosen - x,y:" << active[coalchosen].
↪getXpos() << "," << active[coalchosen].getYpos() << endl;
                                cerr <<" x,y wrap: " << active[coalchosen].getXwrap()
↪<< "," << active[coalchosen].getYwrap() << endl;
                                throw Fatal_Exception("ERROR_MOVE_006: NON FATAL.␣
↪Nwrap not set correctly. Check move programming function.");
                            }
                            if(active[coalchosen].getXpos() != (unsigned long) oldx␣
↪|| active[coalchosen].getYpos() != (unsigned long) oldy || active[coalchosen].
↪getXwrap() != oldxwrap || active[coalchosen].getYwrap() != oldywrap)
                            {
                                cerr << chosen << "," << coalchosen << endl;
                                cerr << "chosen - x,y: " << oldx << "," << oldy <<␣
↪endl << "x, y wrap: " << oldxwrap << "," << oldywrap << endl;
                                cerr << "coalchosen - x,y:" << active[coalchosen].
↪getXpos() << "," << active[coalchosen].getYpos() << endl;
                                cerr <<" x,y wrap: " << active[coalchosen].getXwrap()
↪<< "," << active[coalchosen].getYwrap() << endl;
                                throw Fatal_Exception("ERROR_MOVE_006: NON FATAL.␣
↪Nwrap not set correctly. Check move programming function.");
                            }
                            #endif
                            coalescenceEvent(chosen,coalchosen);
                        }
                        else // debugging only now as the move process has been␣
↪incorportated into calcNewPos().
                        {
                            // for debugging only
                            if(forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,
↪generation)==0)
                            {
                                throw Fatal_Exception(string("ERROR_MOVE_007:␣
↪Dispersal attempted to non-forest. Check dispersal function. Forest cover: " + to_
↪string((long long)forestmap.getVal(oldx,oldy,oldxwrap,oldywrap,generation))));
                            }
                        }
                    }

                }
```

```
            }
            catch(Fatal_Exception& fe)
            {
                cerr << fe.what() << endl;
                exit(-1);
            }
            #ifdef DEBUG
            try
            {
                validationCheck(chosen,112,chosen, coal); // this can probably be␣
↪removed now
            }
            catch(Fatal_Exception &fe)
            {
                cerr << fe.what() << endl;
                cerr << "Location tag: " << location << endl;
                throw Fatal_Exception();
            }
            #endif
            try
            {
                //cout << "check3" << endl;
                runChecks(chosen,coalchosen); // as can this (but there is likely␣
↪minimal performance hit and it does prevent errors).
                // if debug mode is enabled, run a full debug every single time step␣
↪(this will have  a massive performance hit)
                #ifdef debug_mode
                // runs the debug every 10,000 time steps
                if(steps%10000 == 0 )
                {
                    for(int i =0; i<=endactive;i++)
                    {
                        runChecks(i,i);
                    }
                }
                #endif
            }
            catch(Fatal_Exception& fe)
            {
                cerr << fe.what() << endl;
                cerr << "dumping data file..." << endl;
                sqlCreate();
                #ifdef sql_ram
                sqlOutput();
                #endif
                cerr << "done!" << endl;
            }
        }
        // If the simulations finish correctly, output the completed data.
        // Otherwise, pause the simulation and save objects to file.

        if(endactive>1)
        {
            cout << "........out of time!" << endl << "Pausing simulation: add extra␣
↪time or re-run to ensure simulation completion." << endl;
            time(&sim_finish);
            simPause();
            return false;
```

```
        }
        else
        {
            for(unsigned int i=0;i<=endactive;i++)
            {
                data[active[i].getMpos()].speciate();
                data[active[i].getMpos()].setSpec(0);
            }
            if(!bContinueSim)
            {
                cout << "\rBeginning simulations.....done - desired number of␣
↪species achieved!" << endl;
                time(&sim_finish);
                return true;
//              sqlCreate();
            }
            else
            {
                time(&sim_finish);
                cout  << "\rBeginning simulations.....done!
↪                                          "<<endl;
                cout << "Exiting main loop" << endl;
                return true;
//              sqlCreate();
            }
        }
    }


    void expandMap(double generationin)
    {
//      cout << "expanding..." << endl;
//  / loop over the grid first to check how many we need to add
//      cout << "enddata: " << enddata << endl;
//      cout << "endactive: " << endactive << endl;
        unsigned long added_active = 0;
        unsigned long added_data =0;
        for(int i = 0;i< grid.GetCols();i++)
        {
//          cout << i << endl;
            for(int j = 0;j < grid.GetRows();j++)
            {
                if(samplegrid.getVal(i,j))
                {
                    added_active += (long(forestmap.getVal(i,j,0,0,generationin)) -␣
↪grid[j][i].getListsize());
                }
            }
        }
//      cout << "added_data: " << added_data << endl;
//      cout << "added_active: " << added_active << endl;
        added_data = added_active;
        // count all the tips
        for( unsigned int i =0; i < enddata; i++)
        {
            if(samplegrid.getVal(data[i].getXpos(), data[i].getYpos()) && data[i].
↪getXwrap() == 0 && data[i].getYwrap() == 0)
            {
```

```
                // if the lineage isn't yet a tip, we can just make it a tip.
                if(data[i].isTip())
                {
                    added_data ++;
                }
            }
        }
//      cout << "added_data: " << added_data << endl;
//      cout << "added_active: " << added_active << endl;
        // now resize data and active if necessary
        checkSimSize(added_data, added_active);
        // set the lineages that currently exist within sample area to a tip
        for( unsigned int i =1; i < endactive; i++)
        {
//          if(samplegrid.getVal(data[active[i].getMpos()].getXpos(), data[active[i].
↪getMpos()].getYpos()) && data[active[i].getMpos()].getXwrap() == 0 &&␣
↪data[active[i].getMpos()].getYwrap() == 0)
            // TEST new method
            if(samplegrid.getVal(active[i].getXpos(), active[i].getYpos()) &&␣
↪active[i].getXwrap() == 0 && active[i].getYwrap() == 0)
            {
                // if the lineage isn't yet a tip, we can just make it a tip.
                if(!data[active[i].getMpos()].isTip())
                {
                    data[active[i].getMpos()].setGeneration(generationin);
                    data[active[i].getMpos()].setTip(true);
                }
                // if the lineage is already a tip, we must create a new tip to allow␣
↪for the simulation building afterwards.
                else
                {
                    enddata ++;
                    // need to change the species list reference
                    try
                    {
                        grid[active[i].getYpos()][active[i].getXpos()].
↪setSpecies(active[i].getListpos(),i);
                    }
                    catch(runtime_error &re)
                    {
                        cerr << "RP_01" << endl;
                        cerr << re.what() << endl;
                        exit(EXIT_FAILURE);
                    }
                    data[enddata].setup(true,active[i].getXpos(),active[i].getYpos(),
↪active[i].getXwrap(),active[i].getYwrap(),generationin);
                    // Now link the old tip to the new tip
                    data[active[i].getMpos()].setParent(enddata);
                    data[enddata].setIGen(0);
                    data[enddata].setSpec(NR.d01());
                    active[i].setMpos(enddata);
            //      removeOldPos(chosen);
                }
            }
        }

        // add new lineages.
        for( int i = 0;i< grid.GetCols();i++)
```

```
        {
//          cout << i << endl;
            for( int j = 0;j < grid.GetRows();j++)
            {
                if(samplegrid.getVal(i,j))
                {
                    unsigned short mapcover = forestmap.getVal(i,j,0,0,generationin)␣
↪* deme_sample;
                    int num_to_add = mapcover - grid[j][i].getListsize() ;
                    if(num_to_add > 0)
                    {
                        if(mapcover >= grid[j][i].getMaxsize())
                        {
                            grid[j][i].changePercentCover(mapcover);
                        }
                        if( num_to_add + grid[j][i].getListsize() > mapcover)
                        {
                            throw Fatal_Exception("Shouldn't add more than the␣
↪mapcover.");
                        }
                        for(int k = 0; k < num_to_add; k++)
                        {
                            endactive ++;
                            enddata ++;
                            int listpos = 0;
                            try
                            {
                                listpos = grid[j][i].addSpecies(endactive);
                            }
                            catch(out_of_range & oor)
                            {
                                cerr << "k: " << k << endl;
                                cerr <<"num_to_add: " << num_to_add <<endl;
                                cerr <<"mapcover: " << mapcover << endl;
                                cerr << "maxsize: " << grid[j][i].getMaxsize() <<␣
↪endl;
                                cerr << "listsize: " << grid[j][i].getListsize() <<␣
↪endl;
                                cerr << oor.what() << endl;
                                exit(EXIT_FAILURE);
                            }
                            // Add the species to active
                            active[endactive].setup(i,j,0,0,enddata,listpos,1);
                            // Add a tip in the Treenode for calculation of the␣
↪coalescence tree at the end of the simulation.
                            // This also contains the start x and y position of the␣
↪species.
                            data[enddata].setup(true,i,j,0,0,generationin);
                            data[enddata].setSpec(NR.d01());
                        }
                    }
                }
            }
        }
        // double check sizes
        if(enddata > data.size() || endactive > active.size())
        {
            cerr << "Enddata: " << enddata << endl;
```

```
            cerr << "data.size : " << data.size() << endl;
            cerr << "Endactive: " << endactive << endl;
            cerr << "active.size: " << active.size() << endl;
            throw Fatal_Exception("ERROR_MAIN_012: FATAL. Enddata or endactive is␣
↪greater than the size of the relevant object. Programming error likely.");
        }
        if(endactive > startendactive)
        {
            startendactive = endactive;
        }
    }

    unsigned long sortData()
    {
        // Sort and process the species list so that the useful information can be␣
↪extracted from it.
        cout << "Finalising data..." << flush;
        // coalescence finished - process speciation
        // check the data structure
        if(enddata >data.size())
        {
            cerr << "enddata: " << enddata << endl;
            cerr << "data.size(): " << data.size() << endl;
            throw Fatal_Exception("Enddata greater than data size. Programming error␣
↪likely.");
        }
         // Now make sure those left in endactive will definitely speciate.
        for(unsigned int i=1;i<=endactive;i++)
        {
            // check
            data[active[i].getMpos()].setSpec(0.0);
        }
        // Double check speciation events have been counted.
        unsigned long spec_up_to = 0;
        for (unsigned int i =1; i <= enddata;i++)
        {
            if(checkSpeciation(data[i].getSpecRate(), spec, data[i].getGenRate()))
            {
                spec_up_to ++;
                data[i].speciate();
            }
        }
        //cout << "Precount1: " << lineages.countSpecies() << endl;
        //lineages.resetTree();
        // Calculates the lineage data for the minimum speciation rate. // removed␣
↪this process as output data is not required to be as a calculated tree.
        //unsigned long spec_up_to = 1;
        // disabled checks...
        // here we check the data is valid - only required for debugging
        try
        {
            for (unsigned long i = 1 ; i <= enddata ; i ++)
            {
                if ((!(data[i].hasSpeciated()))&&(data[i].getParent()==0&&data[i].
↪getExistance()))
                {
                    throw Main_Exception(string("ERROR_MAIN_004: " +to_string((long␣
↪long)i) +" has not speciated and parent is 0."));
```

```
                }
                if(data[i].isTip() && !samplegrid.getVal(data[i].getXpos(),data[i].
↪getYpos()))
                {
                    cerr << "ERROR_MAIN_014: Tip assignment error. Samplemask is not␣
↪correctly assigning tips. Check samplemask programming." <<endl;
                }
            }

            // here we check the data is valid - alternative validity check.
            for (unsigned long i = 1 ; i <= enddata ; i ++)
            {
                if (!(data[i].hasSpeciated())&&data[i].getExistance())
                {
                    long j = i;
                    while (!(data[j].hasSpeciated()))
                    {
                        j = data[j].getParent();
                        if (j == 0)
                        {
                            throw Main_Exception("ERROR_MAIN_005:0 found in parent␣
↪while following speciation trail.");
                            //string james;
                            //cin >> james;
                        }
                    }
                }
            }
        }
        catch(Main_Exception& me)
        {
            cerr << me.what() << endl;
            cerr << "Returning max possible size (may cause RAM issues)." << endl;
            return data.size();
        }
        cout << "done!" << endl;
        return spec_up_to;
    }

    void outputData()
    {
        unsigned long species_richness = sortData();
        outputData(species_richness);
        return;
    }
    void outputData(unsigned long species_richness)
    {
        // Run the data sorting functions and output the data into the correct format.
        // sort the data
//          bool loopon;
        // Write the data to the first file
        // species richness data
        cout << "Writing results to file..." << flush;
        ofstream out;
        out.precision(10);
        string filename_ab;
        // sprintf (filename_ab, "%s/Data_%i.csv",outdirectory, int(the_task)); //␣
↪altered in v3.0
```

```
        filename_ab = outdirectory + string("/Data_") + to_string((long long)the_
↪task)+"_"+to_string((long long)the_seed) + string(".csv");
        out.open(filename_ab);
        // Simulation fixed variables
        out << "random_seed=," << the_seed << "\n";
        out << "fine map=," << finemapinput<< "\n";
        out << "fine map x y," << finemapxsize << " - " << finemapysize << "\n";
        out << "coarse map=," << coarsemapinput<< "\n";
        out << "coarse map x y," << coarsemapxsize*deme << " - " <<␣
↪coarsemapysize*deme << "\n";
        out << "deme=," << deme << "\n";
        out << "deme_sample=," << deme_sample << "\n";
        out << "spec=," << spec << "\n";
        out << "zfat=," << zfat << "\n";
        out << "maxtime=," << maxtime << "\n\n";
        out << "species_richness=," << species_richness << "\n\n";
        out.close();

        // now we sort the map output and store to another file
        // removed the sparse output format - v3.0
         // this bit is now uneccessary and has been removed.
        /*
        if(false)
        {
            // output data as matrix
            string filename_map;
//              char filename_map[100];
//              sprintf (filename_ab, "%s/Map_dense%i.csv",outdirectory, int(the_
↪task));
            filename_map = outdirectory + string("/Map_dense") + to_string(the_task)+
↪"_"+to_string(the_seed) + string(".csv");

            out.open(filename_map);

            Matrix<unsigned short> toret;
            toret.SetSize(finemapysize,finemapxsize);
            for (long i = 0 ; i < finemapysize; i ++)
            {
                for (long j = 0 ; j < finemapxsize; j ++)
                {
                    toret[i][j] = 0;
                }
            }

            for (unsigned long i = 1 ; i <= enddata ; i ++)
            {
                if (data[i].isTip())
                {
                    // Write data to a toret object.
                    toret[data[i].getYpos()][data[i].getXpos()]++;

                }
            }
            // Write toret object to a file.
            out << toret << "\n";
            out.close();
        }
        //*/
```

```
        cout << "done!                " << endl;
        time(&out_finish);
        #ifdef sql_ram
        sqlOutput();
        #endif
        time(&sim_end);
        writeTimes();
    }

    void writeTimes()
    {
        cout << "Total generations simulated (steps): " << generation << " (" <<␣
↪steps << ")" <<endl;
        cout << "Setup time was " << floor((sim_start - start)/60)<< " minutes " <<␣
↪(sim_start-start)%60<< " seconds" << endl;
        cout << "Simulation time was " <<floor((sim_finish - sim_start)/3600) << "␣
↪hours " << (floor((sim_finish - sim_start)/60)-60*floor((sim_finish - sim_start)/
↪3600)) << " minutes " << (sim_finish-sim_start)%60<< " seconds" << endl;
        cout << "File output and species calculation time was " << floor((out_finish -
↪ sim_finish)/60)<< " minutes " << (out_finish-sim_finish)%60<< " seconds"<< endl;
        cout << "SQL output time was " << floor((sim_end - out_finish)/60)<< "␣
↪minutes " << (sim_end-out_finish)%60<< " seconds" << endl;
         cout << "Total time taken was " << floor((sim_end - start)/3600) << " hours
↪" << (floor((sim_end - start)/60)-60*floor((sim_end - start)/3600)) << " minutes "<
↪< (sim_end-start)%60<< " seconds" << endl;
    }

    void simPause()
    {
        // Completely changed how this sections works - it won't currently allow␣
↪restarting of the simulations, but will dump the data file to memory. - simply␣
↪calls sqlCreate and sqlOutput.
        //sqlCreate();
        //sqlOutput();

        // This function saves the data to 4 files. One contains the main simulation␣
↪parameters, the other 3 contain the simulation results thus far
        // including the grid object, data object and active object.
        cout << "Pausing simulation..." << endl << "Saving data to temp file in " <<␣
↪outdirectory << "/Pause/ ..." << flush;
        ofstream out;
        out.precision(10);
        string file_to_open;
        // Create the pause directory
        string pause_folder = outdirectory + "/Pause/";
        boost::filesystem::path pause_dir(pause_folder);
        if(!boost::filesystem::exists(pause_dir))
        {
            try
            {
                boost::filesystem::create_directory(pause_dir);
            }
            catch(exception &e)
            {
                cerr << "Failure to create " << outdirectory << "/Pause/" << "." <<␣
↪endl;
                cerr << e.what() << endl;
                cerr << "Writing directly to output directory." << endl;
```

```
                pause_folder = outdirectory;
            }
        }
        try
        {
            file_to_open = pause_folder + "Dump_main_" + to_string(the_task) + "_" +␣
↪to_string(the_seed) + ".csv";
            out.open(file_to_open.c_str());
            // Saving the initial data to one file.
            out << enddata << "\n" << seeded  << "\n" << the_seed  << "\n" << the_
↪task  << "\n" << autocorrel_file << "\n" << bAutocorrel  << "\n" << bSpec  << "\n";
            out << finemapinput << "\n" << coarsemapinput << "\n" <<outdirectory <<
↪"\n" << pristinefinemapinput << "\n" << pristinecoarsemapinput << "\n";
            out << dPristine << "\n" << dForestTransform << "\n" << gridxsize << "\n"
↪<< gridysize << "\n" << finemapxsize << "\n" << finemapysize << "\n";
            out << finemapxoffset << "\n" << finemapyoffset << "\n" << coarsemapxsize
↪<< "\n" << coarsemapysize << "\n" << coarsemapxoffset << "\n";
            out << coarsemapyoffset << "\n" << coarsemapscale << "\n" << varimport <<
↪"\n" << start << "\n" << sim_start << "\n" << sim_end << "\n" << now << "\n";
            out << tCheckSpec << "\n" << sim_finish << "\n" << out_finish << "\n" <<␣
↪endactive << "\n" << startendactive << "\n" << maxsimsize << "\n" << steps << "\n";
            out << generation << "\n" << zfat << "\n" << dLvalue << "\n" << maxtime <
↪< "\n" << deme_sample << "\n" << spec << "\n" << lambda << "\n" << deme << "\n";
            out << desired_specnum << "\n" << sqloutname << "\n" << NR << mapvarsin;
            out.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failed to perform main dump to " << pause_folder << endl;
        }
        try
        {
            // Saving the larger data in single files for simpler reading in.
            // Output the grid object
            ofstream out2;
            file_to_open = pause_folder + "Dump_grid_" + to_string(the_task) + "_" +␣
↪to_string(the_seed) + ".csv";
            out2.precision(10);
            out2.open(file_to_open.c_str());
            out2 << grid;
            out2.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failed to perform grid dump to " << pause_folder << endl;
        }
        try
        {
            // Output the active object
            ofstream out3;
            file_to_open = pause_folder + "Dump_active_" + to_string(the_task) + "_"␣
↪+ to_string(the_seed) + ".csv";
            out3.precision(10);
            out3.open(file_to_open.c_str());
            out3 << active;
            out3.close();
```

```
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failed to perform active dump to " << pause_folder << endl;
        }
        try
        {
        // Output the data object
            ofstream out4;
            file_to_open = pause_folder + "Dump_data_" + to_string(the_task) + "_" +␣
↪to_string(the_seed) + ".csv";
            out4.precision(10);
            out4.open(file_to_open.c_str());
            out4 << data;
            out4.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failed to perform data dump to " << pause_folder << endl;
        }
        try
        {
        // Output the data object
            ofstream out4;
            file_to_open = pause_folder + "Dump_map_" + to_string(the_task) + "_" +␣
↪to_string(the_seed) + ".csv";
            out4.precision(10);
            out4.open(file_to_open.c_str());
            out4 << forestmap;
            out4.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failed to perform map dump to " <<  pause_folder << endl;
        }
        cout << "done!" << endl;
        cout << "SQL dump started" << endl;
        time(&out_finish);
        sqlCreate();
        #ifdef sql_ram
        sqlOutput();
        #endif
        cout << "Data dump complete" << endl;
        time(&sim_end);
        writeTimes();
    }

    void simResume(bool cont_sim)
    {
        // Only resume the simulation if there is a simulation to resume from.
        if(!cont_sim)
        {
            return;
        }
        // Loads the data from the files into the relevant objects.
```

```
        cout << "Resuming simulation..." << endl << "Loading data from temp file..." <
↪< flush;
        string file_to_open;
        try
        {
            cout << "\rLoading data from temp file...main..." << flush;
            ifstream in1;
    //      char file_to_open[100];
    //      sprintf(file_to_open,"%s/Pause/Data_%i.csv",outdirectory,int(the_task));
            file_to_open = outdirectory + string("/Pause/Dump_main_") + to_string(the_
↪task)+"_"+to_string(the_seed) + string(".csv");
            in1.open(file_to_open);
            // Reading the initial data
            in1 >> enddata  >> seeded   >> the_seed   >> the_task   >> autocorrel_
↪file  >> bAutocorrel   >> bSpec  ;
            in1 >> finemapinput  >> coarsemapinput  >> outdirectory  >>␣
↪pristinefinemapinput  >> pristinecoarsemapinput ;
            in1 >> dPristine  >> dForestTransform  >> gridxsize  >> gridysize  >>␣
↪finemapxsize  >> finemapysize ;
            in1 >> finemapxoffset  >> finemapyoffset  >> coarsemapxsize  >>␣
↪coarsemapysize  >> coarsemapxoffset ;
            in1 >> coarsemapyoffset  >> coarsemapscale  >> varimport  >> start  >>␣
↪sim_start  >> sim_end  >> now ;
            in1 >> tCheckSpec  >> sim_finish  >> sim_finish  >> endactive  >>␣
↪startendactive  >> maxsimsize  >> steps ;
            unsigned long tempmaxtime;
            in1 >> generation  >> zfat  >> dLvalue  >> tempmaxtime  >> deme_sample  >>
↪ spec  >> lambda  >> deme ;
            in1 >> desired_specnum  >> sqloutname  >> NR >> mapvarsin;
            in1.close();
            //cout << "file closed" << endl;
            try
            {
                if(autocorrel_file == "null")
                {
                    if(bAutocorrel == true)
                    {
                        throw runtime_error("bAutocorrel should not be true");
                    }
                }
                else
                {
                    if(bAutocorrel == false)
                    {
                        throw runtime_error("bAutocorrel should not be false");
                    }
                    vector<string> tmpimport;
                    ConfigOption tmpconfig;
                    tmpconfig.setConfig(autocorrel_file,false);
                    tmpconfig.importConfig(tmpimport);
                    for(unsigned int i = 0;i<tmpimport.size();i++)
                    {
                        autocorrel_times.push_back(stod(tmpimport[i]));
    //                  cout << "t_i: " << autocorrel_times[i] << endl;
                    }
                }
            }
            catch(Config_Exception& ce)
```

```
            {
                cerr << ce.what() << endl;
            }
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failure to import from "  << file_to_open << endl;
        }
        try
        {
            cout << "\rLoading data from temp file...grid..." << flush;
            // Reading the larger data from individual files.
            // Input the grid object
            ifstream in2;
    //      sprintf(file_to_open,"%s/Pause/Data_%i_grid.csv",outdirectory,int(the_
↪task));
            file_to_open = outdirectory + string("/Pause/Dump_grid_") + to_string(the_
↪task)+"_"+to_string(the_seed) + string(".csv");
            in2.open(file_to_open);
            //grid.SetSize(1,400);
            in2 >> grid;
            in2.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failure to import from "  << file_to_open << endl;
        }
        //cout << grid;
        try
        {
            cout << "\rLoading data from temp file...active..." << flush;
            // Input the active object
            ifstream in3;
    //      sprintf(file_to_open,"%s/Pause/Data_%i_active.csv",outdirectory,int(the_
↪task));
            file_to_open = outdirectory + string("/Pause/Dump_active_") + to_
↪string(the_task)+"_"+to_string(the_seed) + string(".csv");
            in3.open(file_to_open);
            in3 >> active;
            in3.close();
        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failure to import from "  << file_to_open << endl;
        }
        // Input the data object
        try
        {
            cout << "\rLoading data from temp file...data..." << flush;
            ifstream in4;
    //      sprintf(file_to_open,"%s/Pause/Data_%i_data.csv",outdirectory,int(the_
↪task));
            file_to_open = outdirectory + string("/Pause/Dump_data_") + to_string(the_
↪task)+"_"+to_string(the_seed) + string(".csv");
            in4.open(file_to_open);
```

```
            in4 >> data;
            in4.close();
            cout << "done!" << endl;

        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failure to import from "  << file_to_open << endl;
        }
        // Input the map object
        try
        {
            cout << "\rLoading data from temp file...map..." << flush;
            ifstream in5;
    //      sprintf(file_to_open,"%s/Pause/Data_%i_data.csv",outdirectory,int(the_
↪task));
            file_to_open = outdirectory + string("/Pause/Dump_map_") + to_string(the_
↪task)+"_"+to_string(the_seed) + string(".csv");
            in5.open(file_to_open);
            in5 >> forestmap;
            in5.close();
            cout << "done!" << endl;

        }
        catch(exception &e)
        {
            cerr << e.what() << endl;
            cerr << "Failure to import from "  << file_to_open << endl;
        }
        cout << "\rLoading data from temp file...done!" << endl;
    }


    void sqlCreate()
    {
        time(&out_finish);
        cout << "Creating SQL database file..." << endl;
        cout << "    Checking for existing folders...." << flush;
        // Create the folder if it doesn't exist
        sqloutname = outdirectory;
        string sqlfolder = outdirectory + "/SQL_data/";
        try
        {
            boost::filesystem::path dir(sqlfolder);
            if(!boost::filesystem::exists(dir))
            {
                cout << "\r    SQL_data folder not found....creating..." << flush;
                if(boost::filesystem::create_directory(sqlfolder))
                {
                    cout << "\r    SQL_data folder created successfully
↪               " << flush;
                }
                else
                {
                    cout << "\r    SQL_data folder not created successfully
↪                   " << endl;
                    throw Main_Exception("ERROR_SQL_012: SQL folder could not be␣
↪created. Check write permissions and that the parent folder is accessible.");
```

```
                }
            }
            else
            {
                cout << "\r    Checking for existing folders....done!        " <<␣
↪flush;
            }
        // create the empty file, deleting if it exists.
            sqloutname += string("/SQL_data/data_")+to_string(the_task)+"_"+to_
↪string(the_seed)+".db";
        }
        catch(Main_Exception& me)
        {
            cerr << me.what() << endl;
            sqloutname = string("data_") + to_string(the_task)+"_"+to_string(the_
↪seed)+".db";
        }
//      remove(sqloutname.c_str());
        cout << "\r    Generating species list....              " << flush;
        // for outputting the full data from the simulation in to a SQL file.
        sqlite3_stmt * stmt;
        char * sErrMsg;
        int rc =0;
        // Open a SQL database in memory. This will be written to disk later.
        // A check here can be done to write to disc directly instead to massively␣
↪reduce RAM consumption
        #ifdef sql_ram
        sqlite3_open(":memory:",&database);
        #endif
        #ifndef sql_ram
        rc = sqlite3_open_v2(sqloutname.c_str(),&database,SQLITE_OPEN_READWRITE |␣
↪SQLITE_OPEN_CREATE, "unix-dotfile");

        #endif
        // Create the command to be executed by adding to the string.
        string all_commands;
        all_commands = "CREATE TABLE SPECIES_LIST (ID int PRIMARY KEY NOT NULL,␣
↪unique_spec INT NOT NULL, xval INT NOT NULL,";
        all_commands += "yval INT NOT NULL, xwrap INT NOT NULL, ywrap INT NOT NULL,␣
↪tip INT NOT NULL, speciated INT NOT NULL, parent INT NOT NULL,existance INT NOT␣
↪NULL,randnum DOUBLE NOT NULL, gen_alive INT NOT NULL,gen_added DOUBLE NOT NULL);";

        // Create the table within the SQL database
        rc = sqlite3_exec(database,all_commands.c_str(),NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            #ifndef sql_ram
            sqlite3_close(database);
//          cerr << "unix-dotfile not working - attempting default SQL VFS method..."
↪<< flush;
            // delete any old database files - this is risky, but there isn't a␣
↪better way of ensuring that the file actually gets created.
            remove(sqloutname.c_str());
            rc = sqlite3_open(sqloutname.c_str(),&database);
            rc = sqlite3_exec(database,all_commands.c_str(),NULL,NULL,&sErrMsg);
            if( rc == SQLITE_OK)
            {
//              cerr << "done! Default unix VFS method functional." << endl;
```

```
            }
            else
            {
                cerr << "Database file creation failed. Check file system." << endl;
                cerr << "Error code: " << rc << endl;
                exit(-1);
            }
            #endif
            if(rc!=SQLITE_OK)
            {
                cerr << "ERROR_SQL_007: Cannot generate in-memory table. Check memory␣
↪database assignment and SQL commands." << endl;
                cerr << "Error code: " << rc << endl;
            }
        }
        // Now create the prepared statement into which we shall insert the values␣
↪from the table
        all_commands = "INSERT INTO SPECIES_LIST (ID,unique_spec,xval,yval,xwrap,
↪ywrap,tip,speciated,parent,existance,randnum,gen_alive,gen_added) VALUES (?,?,?,?,?,
↪?,?,?,?,?,?,?,?,?)";
        sqlite3_prepare_v2(database,all_commands.c_str(),strlen(all_commands.c_str()),
↪&stmt,NULL);

        // Start the transaction
        rc = sqlite3_exec(database,"BEGIN TRANSACTION;",NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            cerr << "ERROR_SQL_008: Cannot start SQL transaction. Check memory␣
↪database assignment and SQL commands." << endl;
        }
        for(unsigned int i =0; i<=enddata;i++)
        {
            sqlite3_bind_int(stmt,1,i);
            sqlite3_bind_int(stmt,2,data[i].getSpeciesID());
            sqlite3_bind_int(stmt,3,data[i].getXpos());
            sqlite3_bind_int(stmt,4,data[i].getYpos());
            sqlite3_bind_int(stmt,5,data[i].getXwrap());
            sqlite3_bind_int(stmt,6,data[i].getYwrap());
            sqlite3_bind_int(stmt,7,data[i].isTip());
            sqlite3_bind_int(stmt,8,data[i].hasSpeciated());
            sqlite3_bind_int(stmt,9,data[i].getParent());
            sqlite3_bind_int(stmt,10,data[i].getExistance());
            sqlite3_bind_double(stmt,11,data[i].getSpecRate());
            sqlite3_bind_int(stmt,12,data[i].getGenRate());
            sqlite3_bind_double(stmt,13,data[i].getGeneration());
            sqlite3_step(stmt);
            sqlite3_clear_bindings(stmt);
            sqlite3_reset(stmt);
        }


        cout << "\r    Executing SQL commands...." << flush;
        // execute the command and close the connection to the database
        rc = sqlite3_exec(database,"END TRANSACTION;",NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            cerr << "ERROR_SQL_008: Cannot complete SQL transaction. Check memory␣
↪database assignment and SQL commands. Ensure SQL statements are properly cleared." <
↪< endl;
```

```
            cerr << "Error code: " << rc << endl;
            // try again
            int i =0;
            while((rc!=SQLITE_OK && rc != SQLITE_DONE )&& i < 10)
            {
                sleep(1);
                i ++;
                rc = sqlite3_exec(database,"END TRANSACTION;",NULL,NULL,&sErrMsg);
                cerr << "Attempt " << i << " failed..." << endl;
                cerr << "ERROR_SQL_008: Cannot complete SQL transaction. Check memory␣
↪database assignment and SQL commands. Ensure SQL statements are properly cleared." <
↪< endl;
            }
        }
        // Need to finalise the statement
        rc = sqlite3_finalize(stmt);
        if(rc!=SQLITE_OK)
        {
            cerr << "ERROR_SQL_008: Cannot complete SQL transaction. Check memory␣
↪database assignment and SQL commands. Ensure SQL statements are properly cleared." <
↪< endl;
            cerr << "Error code: " << rc << endl;
        }
        // Vacuum the file so that the file size is reduced (reduces by around 3%)
        rc = sqlite3_exec(database,"VACUUM;",NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            cerr << "ERROR_SQL_014: Cannot vacuum the database. Error message: " <<␣
↪sErrMsg << endl;
        }

        // Now additionally store the simulation parameters (extremely useful data)
        string to_execute = "CREATE TABLE SIMULATION_PARAMETERS (job_num INT PRIMARY␣
↪KEY not null, job_type INT NOT NULL,";
        to_execute += "output_dir TEXT NOT NULL, spec_rate DOUBLE NOT NULL, zfat␣
↪DOUBLE NOT NULL,dispersal DOUBLE NOT NULL, deme INT NOT NULL, ";
        to_execute += "sample_size DOUBLE NOT NULL, max_time DOUBLE NOT NULL, lambda␣
↪DOUBLE NOT NULL, min_num_species INT NOT NULL, forest_change_param DOUBLE NOT NULL,
↪";
        to_execute += "time_since_pristine DOUBLE NOT NULL, time_config_file TEXT NOT␣
↪NULL,coarse_map_file TEXT NOT NULL, coarse_map_x INT NOT NULL, coarse_map_y INT NOT␣
↪NULL,";
        to_execute += "coarse_map_x_offset INT NOT NULL, coarse_map_y_offset INT NOT␣
↪NULL, coarse_map_scale DOUBLE NOT NULL, fine_map_file TEXT NOT NULL, fine_map_x INT␣
↪NOT NULL,";
        to_execute += "fine_map_y INT NOT NULL, fine_map_x_offset INT NOT NULL, fine_
↪map_y_offset INT NOT NULL, grid_x INT NOT NULL, grid_y INT NOT NULL,";
        to_execute += "pristine_coarse_map TEXT NOT NULL, pristine_fine_map TEXT NOT␣
↪NULL);";
        rc = sqlite3_exec(database,to_execute.c_str(),NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            cerr << "ERROR_SQL_008: Cannot start SQL transaction. Check memory␣
↪database assignment and SQL commands." << endl;
            cerr << "Error code: " << rc << endl;
        }
        to_execute = "INSERT INTO SIMULATION_PARAMETERS VALUES(" + to_string((long␣
↪long)the_task) + "," + to_string((long long)the_seed) + ",'" + outdirectory+ "'," +␣
↪boost::lexical_cast<std::string>((long double) spec) + "," + to_string((long␣
↪double)zfat) + ",";
```

```
        to_execute += to_string((long double)dLvalue) + "," + to_string((long␣
↪long)deme) + "," + to_string((long double)deme_sample) + "," + to_string((long␣
↪double)maxtime) + "," + to_string((long double)lambda) + "," + to_string((long␣
↪long)desired_specnum) + "," + to_string((long double)mapvarsin.dForestTransform) +
↪",";
        to_execute += to_string((long double)mapvarsin.dPristine) + ",'" + mapvarsin.
↪autocorrel_file + "','" + coarsemapinput + "'," + to_string((long long)␣
↪coarsemapxsize) + "," + to_string((long long)coarsemapysize) + "," + to_
↪string((long long) coarsemapxoffset) + ",";
        to_execute += to_string((long long)coarsemapyoffset) + "," + to_string((long␣
↪long)coarsemapscale) + ",'" + finemapinput + "'," +  to_string((long␣
↪long)finemapxsize) + "," + to_string((long long)finemapysize) + "," +  to_
↪string((long long)finemapxoffset) + "," + to_string((long long)finemapyoffset) + ",
↪";
        to_execute += to_string((long long)gridxsize) +","+ to_string((long␣
↪long)gridysize) + ",'" + pristinecoarsemapinput + "','" + pristinefinemapinput +"');
↪";
        rc = sqlite3_exec(database,to_execute.c_str(),NULL,NULL,&sErrMsg);
        if(rc!=SQLITE_OK)
        {
            cout << to_execute << endl;
            cout << boost::lexical_cast<std::string>((double) spec) << endl;
            cout << spec << endl;
            cerr << "ERROR_SQL_008: Cannot start SQL transaction. Check memory␣
↪database assignment and SQL commands." << endl;
            cerr << "Error code: " << rc << endl;
        }
        cout << "done!" << endl;
    }

    void sqlOutput()
    {
        #ifdef sql_ram
        // open connection to the database file
        remove(sqloutname.c_str());
        cout << "\r    Writing to " << sqloutname << " ....     " << flush;
        int rc = sqlite3_open_v2(sqloutname.c_str(),&outdatabase,SQLITE_OPEN_
↪READWRITE | SQLITE_OPEN_CREATE,"unix-dotfile");
        if(rc!=SQLITE_OK && rc != SQLITE_DONE)
        {
            int i =0;
            while((rc!=SQLITE_OK && rc != SQLITE_DONE) && i < 10)
            {
                i++;
                sleep(1);
                rc = sqlite3_open_v2(sqloutname.c_str(),&outdatabase,SQLITE_OPEN_
↪READWRITE | SQLITE_OPEN_CREATE,"unix-dotfile");
//              cerr << "Attempt " << i << " failed..." << endl;
            }
            // Attempt different opening method if the first fails.
            int j = 0;
            while((rc!=SQLITE_OK && rc != SQLITE_DONE) && j < 10)
            {
                j++;
                sleep(1);
                rc = sqlite3_open(sqloutname.c_str(),&outdatabase);
//              cerr << "Attempt " << i << " failed..." << endl;
            }
```

```
            if(rc != SQLITE_OK && rc != SQLITE_DONE)
            {
                cerr <<  "ERROR_SQL_010: SQLite database file could not be opened.␣
↪Check the folder exists and you have write permissions. (REF1) Error code: " << rc <
↪< endl;
                cerr << "Attempted call " << max(i,j) << " times" << endl;
            }
        }
        // create the backup object to write data to the file from memory.
        sqlite3_backup *backupdb;
        backupdb = sqlite3_backup_init(outdatabase,"main",database,"main");
        if(!backupdb)
        {
            cerr << "ERROR_SQL_011: Could not write to the backup database. Check the␣
↪file exists." << endl;
        }
        // Perform the backup
        rc = sqlite3_backup_step(backupdb,-1);
        if(rc!=SQLITE_OK && rc != SQLITE_DONE)
        {
//          cerr <<  "ERROR_SQL_010: SQLite database file could not be opened. Check␣
↪the folder exists and you have write permissions. (REF2) Error code: " << rc <<␣
↪endl;
            // try again
            int i =0;
            while((rc!=SQLITE_OK && rc != SQLITE_DONE) && i < 10)
            {
                i++;
                sleep(1);
                rc = sqlite3_backup_step(backupdb,-1);
//              cerr << "Attempt " << i << " failed..." << endl;
            }
            if(rc != SQLITE_OK && rc != SQLITE_DONE)
            {
                cerr <<  "ERROR_SQL_010: SQLite database file could not be opened.␣
↪Check the folder exists and you have write permissions. (REF3) Error code: " << rc <
↪< endl;
                cerr << "Attempted call " << i << " times" << endl;
            }
        }
        sqlite3_backup_finish(backupdb);
//      sqlite3_exec(database,"VACUUM;",NULL,NULL,&sErrMsg);
        sqlite3_close(database);
        sqlite3_close(outdatabase);
        cout << "\r    Writing to " << sqloutname << " ....  done!                " <<␣
↪endl;
        #endif
    }

    void applySpecRates(double sr, double t)
    {
        if(!tl.hasSetMem())
        {
            tl.setDatabase(database);
        }
        //cout << "Precount: " << tl.countSpecies() << endl;
        tl.setGeneration(t);
        tl.resetTree();
```

```
        tl.createDatabase(sr);

        #ifdef record_space
        tl.recordSpatial();
        #endif
    }

    void applySpecRates(double sr)
    {
        applySpecRates(sr, 0.0);
    }


};

#endif // TREE
```

Includes

- Config.h (*File Config.h*)
- Datapoint.h (*File Datapoint.h*)
- Fattaildeviate.h (*File Fattaildeviate.h*)
- Map.h (*File Map.h*)
- Matrix.h (*File Matrix.h*)
- SpeciesList.h (*File SpeciesList.h*)
- Treelist.h (*File Treelist.h*)
- Treenode.h (*File Treenode.h*)
- algorithm
- boost/filesystem.hpp
- cmath
- cstring
- ctime
- fstream
- iomanip
- iostream
- math.h
- sqlite3.h
- stdexcept
- stdio.h
- string
- time.h
- unistd.h

- vector

Included By

- *File Setup.h*

Classes

- *Class Tree*

Defines

- *Define sql_ram*

File Treelist.cpp

Definition (`Treelist.cpp`)

Program Listing for File Treelist.cpp

- Return to documentation for *File Treelist.cpp*

```
// This file is part of NECSim project which is released under BSD-3 license.
// See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause)␣
↪for full license details.

//#define use_csv
#include "Treelist.h"
void Treelist::setList(Row<Treenode>& l)
{
    nodes = l;
}

void Treelist::setDatabase(sqlite3* dbin)
{
    if(!bMem)
    {
        database = dbin;
    }
    else
    {
        throw SpeciesException("ERROR_SPEC_002: Attempt to set database - database␣
↪link has already been set");
    }
    bMem = true;  // this just specifies that the database has been created in memory.
}

bool Treelist::hasSetMem()
{
    return bMem;
}
```

```
double Treelist::getMinimumSpeciation()
{
    return min_spec_rate;
}

void Treelist::importSamplemask(string sSamplemask)
{
    if(!bSample)
    {
        samplemaskfile = sSamplemask;
        if(sSamplemask != "null")
        {
            samplemask.SetSize(grid_y_size, grid_x_size);
            //              cout << "importing samplemask: " << sSamplemask << endl;
            samplemask.setIsNull(false);
            samplemask.import(sSamplemask);
            //              Matrix<unsigned short> tempsampmask;
            //              tempsampmask.SetSize(samplemask.GetRows(),samplemask.
↪GetCols());
            //              tempsampmask.importCsv(sSamplemask);
            unsigned long total = 0;
            for(int i = 0; i < samplemask.GetCols(); i++)
            {
                for(int j = 0; j < samplemask.GetRows(); j++)
                {
                    if(samplemask[j][i])
                    {
                        total++;
                        // Check
                        if(!samplemask.getVal(i, j))
                        {
                            cerr << "BUG!" << endl;
                            exit(1);
                        }
                    }
                }
            }
            cout << "Sampling " << total << " cells." << endl;
        }
        else
        {
            samplemask.setIsNull(true);
#ifdef DEBUG
            cout << "Sampling all areas." << endl;
#endif
        }
        bSample = true;
    }
}

unsigned long Treelist::countSpecies()
{
    unsigned int precount = 0;
    for(unsigned int i = 1; i < nodes.size(); i++)
    {
        if(nodes[i].hasSpeciated())
        {
            precount++;
```

```
        }
    }
    // cout << "Precount: " << precount << endl;
    return (precount);
}

bool Treelist::doubleCompare(double d1, double d2, double epsilon)
{
    return (abs(float(d1 - d2)) < epsilon);
}

unsigned long Treelist::calcSpecies(double s)
{
    resetTree();
    if(!bSample)
    {
#ifdef DEBUG
        cerr << "No samplemask imported. Defaulting to null." << endl;
#endif
        importSamplemask("null");
    }
    //      cout << "listsize: " << nodes.size() << endl;
    dSpecRate = s;
    unsigned long iSpecCount = 0;  // start at 2 because the last species has been␣
↪burnt already.
    // check that tips exist within the spatial and temporal frame of interest.
    for(unsigned int i = 1; i < nodes.size(); i++)
    {
        if(nodes[i].isTip() && !samplemask.getVal(nodes[i].getXpos(), nodes[i].
↪getYpos()))
        {
            cout << "Outside sample zone: " << i << " (" << nodes[i].getXpos() << ","
↪<< nodes[i].getYpos()
                 << ")" << endl;
            exit(EXIT_FAILURE);
        }
        if(nodes[i].getParent() >= nodes.size())
        {
            cerr << "i: " << endl;
            cerr << "parent: " << nodes[i].getParent() << endl;
            cerr << "size: " << nodes.size() << endl;
            throw SpeciesException("ERROR_SQL_017: The parent is outside the size of␣
↪the the data object. Bug "
                                   "in expansion of data structures or object set up␣
↪likely.");
        }
        if(nodes[i].isTip() && samplemask.getVal(nodes[i].getXpos(), nodes[i].
↪getYpos()) &&
           doubleCompare(nodes[i].getGeneration(), generation, 0.0001))
        {
            // if they exist exactly in the generation of interest.
            nodes[i].setExistance(true);
        }
        else
        {
            nodes[i].setExistance(false);
        }
        // Calculate if speciation occured at any point in the lineage's branch
```

```
//      double maxret ;
//      if(nodes[i].getGenRate() == 0)
//      {
//          maxret = 1;
//      }
//      else
//      {
//          maxret = nodes[i].getGenRate();
//      }
        if(checkSpeciation(nodes[i].getSpecRate(), s, nodes[i].getGenRate()))
        {
            iSpecCount++;
            nodes[i].speciate();
            //              nodes[i].burnSpecies(iSpecCount);
        }
    }
    //      cout << "check2: " << nodes[1].getExistance() << endl;
    //      bool check = nodes[1].getGeneration() == generation;
    //      cout << "point: " << samplemask.getVal(nodes[1].getXpos(),nodes[1].
↪getYpos()) << endl;
    //      cout << "tip: " << nodes[1].isTip() << endl;
    //      cout << "gen: " << check << endl;
    // cout << "iSpecCount: " << iSpecCount << endl;
    // now continue looping to calculate species identities for lineages given the␣
↪new speciation probabilities.
    bool bSorter = true;
    while(bSorter)
    {
        bSorter = false;
        for(unsigned int i = 1; i < nodes.size(); i++)
        {
            // check if any parents exist
            if(!nodes[nodes[i].getParent()].getExistance() && nodes[i].getExistance()␣
↪&&
               !nodes[i].hasSpeciated())
            {
                bSorter = true;
                nodes[nodes[i].getParent()].setExistance(true);
            }
        }
    }
    //      cout << "2" << endl;
    // Loop to check that no lineages have a parent that is a tip. If they do, set␣
↪their existence to false
    // We then need to continue looping to complete
    //      bool loop = true;
    //      cout << "testloop" << endl;
    //      while(loop)
    //      {
    //          for (unsigned int i =1; i<=nodes.size(); i++)
    //          {
    //              loop = false;
    //              if(nodes[i].getExistance() && nodes[nodes[i].getParent()].
↪getExistance() &&
    //nodes[nodes[i].getParent()].isTip() && nodes[nodes[i].getParent()].
↪getGeneration() <= (generation+0.0001))
    //              {
    //                  nodes[i].setExistance(false);
```

```
    //                  nodes[i].setGeneration(nodes[nodes[i].getParent()].
↪getGeneration());
    //                  loop = true;
    //              }
    //              if(nodes[i].getExistance() && nodes[nodes[i].getParent()].
↪getGeneration() <= (generation + 0.0001) &&
    //              {
    //                  loop = true;
    //                  nodes[i].setExistance(false);
    //                  nodes[i].setGeneration(nodes[nodes[i].getParent()].
↪getGeneration());
    //              }
    //
    //              if(nodes[i].getParent()==0 && nodes[i].getExistance() && !loop &&
    //              {
    //                  cout << "dSpec: " << nodes[i].getSpecRate() <<endl;
    //                  cout << "tip: " << nodes[i].isTip() << endl;
    //                  cout << "i: " << i << endl;
    //                  cout << "parent: " << nodes[i].getParent() << endl;
    //                  cout << "parent_id: " << nodes[nodes[i].getParent()].
↪getSpeciesID() << endl;
    //                  cout <<"existance: " << nodes[i].getExistance() << endl;
    //                  cout << "speciated: " << nodes[i].hasSpeciated() << endl;
    //                  cout << "nodes.size: " << nodes.size() << endl;
    //                  throw SpeciesException("ERROR_SPEC_001: Incorrect setting of␣
↪speciation rate, as individual exists but
    //has no species identity."); // this needs explaining properly at some point.
    //              }
    //          }
    //      }
    iSpecCount = 0;
    // Now loop again, creating a new species for each species that actually exists.
    for(unsigned int i = 1; i < nodes.size(); i++)
    {
        if(nodes[i].getExistance() && nodes[i].hasSpeciated())
        {
            iSpecCount++;
            nodes[i].burnSpecies(iSpecCount);
        }

        //          else if(nodes[i].getExistance()&&nodes[i].hasSpeciated())
        //          {
        //              cout << "i: " << i << endl;
        //              cout << "id: " << nodes[i].getSpeciesID() << endl;
        //          }
    }
    // cout << "iSpecCount: " << iSpecCount << endl;
    //      cout << "3" << endl;

    // now loop to correctly assign each species id
    bool loopon = true;
    while(loopon)
    {
        //          cout << "while:1" << endl;
        // These variables are no longer required
        /*long tempdebug = 0;
        long tempdebug1 = 0;*/
```

```
        loopon = false;
        for(unsigned long i = (nodes.size()) - 1; i > 0; i--)  // if we start at the␣
↪end of the loop and work
                                                               // backwards, we␣
↪should remove some of the repeat
                                                               // speciation events.
        {
            //              cout << i << endl;
            if(nodes[i].getSpeciesID() == 0 && nodes[i].getExistance())
            {
                //                  cout << i << endl;
                loopon = true;
                // tempdebug ++;
                nodes[i].burnSpecies(nodes[nodes[i].getParent()].getSpeciesID());

                if(nodes[nodes[i].getParent()].getSpeciesID() == 0 &&
                   doubleCompare(nodes[i].getGeneration(), generation, 0.001))
                {
                    cout << "dSpec: " << nodes[i].getSpecRate() << endl;
                    cout << "tip: " << nodes[i].isTip() << endl;
                    cout << "i: " << i << endl;
                    cout << "parent: " << nodes[i].getParent() << endl;
                    cout << "parent_id: " << nodes[nodes[i].getParent()].
↪getSpeciesID() << endl;
                    cout << "existance: " << nodes[i].getExistance() << endl;
                    cout << "speciated: " << nodes[i].hasSpeciated() << endl;
                    cout << "generation: " << nodes[i].getGeneration() << endl;
                    cout << "Potential parent ID error - press any key to continue: "
↪<< flush;
                    string test;
                    cin >> test;
                    //                      cout << "i: " << i << endl;
                }
            }
        }
    }
    //      cout << "4" << endl;

    // count the number of species that have been created
    iSpecCount = 0;
    for(unsigned int i = 0; i < nodes.size(); i++)
    {
        if(nodes[i].hasSpeciated() && nodes[i].getExistance() &&
           nodes[i].getSpeciesID() !=
               0)  //&&nodes[i].getExistance()) // count all speciation events, not␣
↪just the ones that exist!
        {
            iSpecCount++;
        }
        if(nodes[i].getExistance() && nodes[i].getSpeciesID() == 0)
        {
            cerr << "ERROR_SQL_006: Species exists but hasn't been assigned a species␣
↪ID. Check the tree "
                    "speciation calculations."
                 << endl;
        }
    }
    iSpecies = iSpecCount;
```

```
    //      cout << "iSpecies: " << iSpecies << endl;
    return iSpecCount;
}

void Treelist::calcSpeciesAbundance()
{
    rOut.SetRowSize(iSpecies + 1);
    //      cout << "iSpecies: " << iSpecies << endl;
    for(unsigned int i = 0; i < rOut.size(); i++)
    {
        rOut[i] = 0;
    }
    for(unsigned int i = 1; i < nodes.size(); i++)
    {
        if(nodes[i].isTip() && doubleCompare(nodes[i].getGeneration(), generation, 0.
↪0001) &&
           nodes[i].getExistance())
        {
            if(!samplemask.getVal(nodes[i].getXpos(), nodes[i].getYpos()) &&
               doubleCompare(nodes[i].getGeneration(), generation, 0.0001))
            {
                cerr << "x,y " << nodes[i].getXpos() << ", " << nodes[i].getYpos() <<␣
↪endl;
                cerr << "tip: " << nodes[i].isTip() << " Existance: " << nodes[i].
↪getExistance()
                     << " samplemask: " << samplemask.getVal(nodes[i].getXpos(),␣
↪nodes[i].getYpos())
                     << endl;
                cerr << "ERROR_SQL_005: Tip doesn't exist. Something went wrong␣
↪either in the import or "
                        "main simulation running."
                     << endl;
            }
            // The line that counts the number of individuals
            rOut[nodes[i].getSpeciesID()]++;
            if(nodes[i].getSpeciesID() == 0 && samplemask.getVal(nodes[i].getXpos(),␣
↪nodes[i].getYpos()) &&
               doubleCompare(nodes[i].getGeneration(), generation, 0.0001))
            {
                cerr << "x,y " << nodes[i].getXpos() << ", " << nodes[i].getYpos() <<␣
↪endl;
                cout << "generation (point,required): " << nodes[i].getGeneration() <
↪< ", " << generation
                     << endl;
                unsigned long p = nodes[i].getParent();
                cout << "samplemasktest: " << samplemask.getTestVal(nodes[i].
↪getXpos(), nodes[i].getYpos())
                     << endl;
                cout << "samplemask: " << samplemask.getVal(nodes[i].getXpos(),␣
↪nodes[i].getYpos()) << endl;
                cout << "parent (tip, exists, generations): " << nodes[p].isTip() <<
↪", "
                     << nodes[p].getExistance() << ", " << nodes[p].getGeneration() <
↪< endl;
                cout << "species id zero - i: " << i << " parent: " << nodes[i].
↪getParent()
                     << " dSpec: " << nodes[i].getSpecRate() << "has speciated: " <<␣
↪nodes[i].hasSpeciated()
```

```
                    << endl;
                throw runtime_error("Fatal, exiting program.");
            }
        }
    }
}

void Treelist::resetTree()
{
    for(unsigned int i = 0; i < nodes.size(); i++)
    {
        nodes[i].qReset();
    }
    // Speciate the final lineages.
    //      nodes[r].speciate();
    //      nodes[r].burnSpecies(1);
    //      nodes[r].setExistance(true);
}

void Treelist::detectDimensions(string db)
{
    sqlite3* tmpdb;
    int rc = sqlite3_open_v2(db.c_str(), &tmpdb, SQLITE_OPEN_READWRITE, "unix-dotfile
↪");
    string to_exec = "SELECT MAX(xval),MAX(yval) FROM SPECIES_LIST;";
    sqlite3_stmt* stmt;
    rc = sqlite3_prepare_v2(tmpdb, to_exec.c_str(), strlen(to_exec.c_str()), &stmt,␣
↪NULL);
    unsigned int xvalmax, yvalmax;
    rc = sqlite3_step(stmt);
    xvalmax = sqlite3_column_int(stmt, 0) + 1;
    yvalmax = sqlite3_column_int(stmt, 1) + 1;
    samplemask.SetSize(xvalmax, yvalmax);
    //      cout << "xvalmax: " << xvalmax << endl;
    //      cout << "yvalmax: " << xvalmax << endl;
    // close the old statement
    rc = sqlite3_finalize(stmt);
    if(rc != SQLITE_OK && rc != SQLITE_DONE)
    {
        cerr << "rc: " << rc << endl;
        throw SpeciesException("Could not detect dimensions");
    }
}

void Treelist::importData(string inputfile)
{
    cout << "Beginning data import..." << flush;
    //  int res = sqlite3_create_function(dbRef, "SPEC_CALC", 2, SQLITE_UTF8, NULL, &
↪sqlite_calc_speciation, NULL,
    //NULL);
    // open the database objects
    sqlite3* outdatabase;
    sqlite3_backup* backupdb;
    sqlite3_stmt* stmt;
    // open one db in memory and one from the file.
    int o1 = sqlite3_open(":memory:", &database);
    int o2 = -1;
    // Check the outdatabase exists
```

```
    if(!boost::filesystem::exists(inputfile))
    {
        throw SpeciesException("ERROR_SQL_015: FATAL. Source file does not exist.");
    }
    o2 = sqlite3_open_v2(inputfile.c_str(), &outdatabase, SQLITE_OPEN_READWRITE,
↪"unix-dotfile");
    // cout << "o2: " << o2 << endl;
    if(o2 != SQLITE_OK && o2 != SQLITE_DONE)
    {
        throw SpeciesException("ERROR_SQL_002: FATAL. Source file cannot be opened.");
    }
    if(sqlite3_errcode(database) != 0)
    {
        cerr << "Can't open in-memory database. Writing to file instead (this will be␣
↪slower)." << endl;
        bMem = false;
        sqlite3_close(database);
        sqlite3_close(outdatabase);
        int rc = sqlite3_open_v2(inputfile.c_str(), &database, SQLITE_OPEN_READWRITE,
↪"unix-dotfile");
        if(rc != SQLITE_OK)
        {
            throw SpeciesException("ERROR_SQL_002: FATAL. Source file cannot be␣
↪opened.");
        }
    }
    else
    {
        bMem = true;
        // copy the db from file into memory.
        backupdb = sqlite3_backup_init(database, "main", outdatabase, "main");
        int rc = sqlite3_backup_step(backupdb, -1);
        // Revert to different VFS file opening method if the backup hasn't started␣
↪properly.
        // Two different versions will be attempted before an error will be thrown.
        // A different way of assigning the VFS method and opening the file correctly␣
↪could be implemented later.
        // Currently "unix-dotfile" works for HPC runs and "unix" works for PC runs.
        if(rc != SQLITE_DONE && rc != SQLITE_OK)
        {
            sqlite3_close(outdatabase);
            sqlite3_open(inputfile.c_str(), &outdatabase);
            backupdb = sqlite3_backup_init(database, "main", outdatabase, "main");
        }
        rc = sqlite3_backup_finish(backupdb);
        //          cout << "rc: " << rc << endl;
        if(rc != SQLITE_DONE && rc != SQLITE_OK)
        {
            throw SpeciesException("ERROR_SQL_002: FATAL. Source file cannot be␣
↪opened.");
            sqlite3_close(database);
            sqlite3_close(outdatabase);
        }
        sqlite3_close(outdatabase);
    }
    // Now find out the max size of the list, so we have a count to work from
    string count_command = "SELECT MAX(ID) FROM SPECIES_LIST;";
    sqlite3_prepare_v2(database, count_command.c_str(), strlen(count_command.c_str()),
↪ &stmt, NULL);
```

```
    unsigned int datasize;
    // skip first row (should be blank)
    sqlite3_step(stmt);
    datasize = sqlite3_column_int(stmt, 0);
    //      cout << "datasize: " << datasize << endl;
    // close the old statement
    sqlite3_finalize(stmt);

    // Create db query
    string all_commands = "SELECT * FROM SPECIES_LIST;";
    sqlite3_prepare_v2(database, all_commands.c_str(), strlen(all_commands.c_str()), &
↪stmt, NULL);
    nodes.SetRowSize(datasize + 1);
    //      cout << endl << "size: " << datasize << endl;
    //  sqlite3_exec(database,"BEGIN TRANSACTION;",NULL,NULL,NULL);
    // Check that the file opened correclty.
    if(o1 != SQLITE_OK || o2 != SQLITE_OK)
    {
        throw SpeciesException("ERROR_SQL_002: FATAL. Source file cannot be opened.");
    }
    sqlite3_step(stmt);
    // Copy the data across to the Treenode data structure.
    // For storing the number of ignored lineages so this can be subtracted off the␣
↪parent number.
    unsigned long ignored_lineages = 0;
    for(unsigned int i = 1; i <= datasize; i++)
    {
        unsigned long species_id = sqlite3_column_int(stmt, 1);
        //      cout << species_id << endl;
        long xval = sqlite3_column_int(stmt, 2);
        long yval = sqlite3_column_int(stmt, 3);
        long xwrap = sqlite3_column_int(stmt, 4);
        long ywrap = sqlite3_column_int(stmt, 5);
        bool tip = bool(sqlite3_column_int(stmt, 6));
        bool speciation = bool(sqlite3_column_int(stmt, 7));
        unsigned long parent = sqlite3_column_int(stmt, 8);
        unsigned long iGen = sqlite3_column_int(stmt, 11);
        bool existance = bool(sqlite3_column_int(stmt, 9));
        double dSpec = sqlite3_column_double(stmt, 10);
        long double generationin = sqlite3_column_double(stmt, 12);
        //          cout << xval << ", " << yval << endl;
        if(tip && !samplemask.getVal(xval, yval) && generationin > generation &&␣
↪false)
        {
            ignored_lineages++;
            sqlite3_step(stmt);
        }
        else
        {
            // the -1 is to ensure that the list includes all lineages, but fills the␣
↪output from the beginning
            unsigned int index = i - 1 - ignored_lineages;
            nodes[index].setup(tip, xval, yval, xwrap, ywrap, generationin);
            nodes[index].burnSpecies(species_id);
            nodes[index].setSpec(dSpec);
            nodes[index].setExistance(existance);
            nodes[index].setIGen(iGen);
            nodes[index].setParent(parent - ignored_lineages);
```

```
            if(index == parent && parent != 0)
            {
                cerr << " i: " << index << " parent: " << parent << endl;
                cerr << "ERROR_SQL_001: Import failed as parent is self. Check import␣
↪function." << endl;
            }
            nodes[index].setSpeciation(speciation);
            sqlite3_step(stmt);
            if(parent < index && !speciation)
            {
                cerr << "parent: " << parent << " index: " << index << endl;
                cerr << "Parent before index error. Check program." << endl;
            }
        }
    }
    // Now we need to blank all objects
    sqlite3_finalize(stmt);
    // Now read the useful information from the SIMULATION_PARAMETERS table
    try
    {
        sqlite3_stmt* stmt2;
        string sql_parameters = "SELECT spec_rate, grid_x,grid_y FROM SIMULATION_
↪PARAMETERS;";
        int rc = sqlite3_prepare_v2(database, sql_parameters.c_str(), strlen(sql_
↪parameters.c_str()), &stmt2, NULL);
        if(rc != SQLITE_DONE && rc != SQLITE_OK)
        {
            sqlite3_close(database);
            sqlite3_close(outdatabase);
            throw SpeciesException("ERROR_SQL_020: FATAL. Could not open simulation␣
↪parameters.");
            //              exit(EXIT_FAILURE);
        }
        sqlite3_step(stmt2);
        min_spec_rate = sqlite3_column_double(stmt2, 0);
        //          cout << sql_parameters << endl;
        //          cout << sqlite3_column_double(stmt2,0) << endl;
        //          cout << sqlite3_column_int(stmt2,2) << endl;
        grid_x_size = sqlite3_column_int(stmt2, 1);
        grid_y_size = sqlite3_column_int(stmt2, 2);
        //          cout << "min: " << min_spec_rate << endl << "x,y: " << grid_x_
↪size << "," << grid_y_size <<
        //endl;
        sqlite3_step(stmt2);
        sqlite3_finalize(stmt2);
        cout << "\rBeginning data import..."
             << "done             " << endl;
    }
    catch(exception& er)
    {
        throw SpeciesException(er.what());
    }
}

void Treelist::setGeneration(long double generationin)
{
    generation = generationin;
}
```

```
void Treelist::createDatabase(double s)
{
    cout << "Calculating tree structure..." << flush;
    if(s < min_spec_rate && s != min_spec_rate)
    {
        cerr << "s: " << s << endl;
        cerr << "min_spec: " << min_spec_rate << endl;
        throw SpeciesException(
            "ERROR_SQL_018: Speciation rate is less than the minimum speciation rate.␣
↪Skipping.");
    }
    else
    {
        unsigned long nspec = calcSpecies(s);
        calcSpeciesAbundance();
        cout << "done!" << endl;
        cout << "Number of species: " << nspec << endl;
        cout << "Generating new SQL table for speciation rate " << s << "..." <<␣
↪flush;
        string table_command = "CREATE TABLE IF NOT EXISTS SPECIES_ABUNDANCES (ID␣
↪TEXT PRIMARY KEY NOT NULL, "
                               "species_id INT NOT NULL, speciation_rate DOUBLE NOT␣
↪NULL, no_individuals INT NOT "
                               "NULL,generation DOUBLE NOT NULL);";
        int rc = sqlite3_exec(database, table_command.c_str(), NULL, NULL, NULL);
        if(rc != SQLITE_OK)
        {
            throw SpeciesException("ERROR_SQL_002b: Could not create SPECIES_
↪ABUNDANCES table.");
        }

        sqlite3_stmt* stmt;
        table_command = "INSERT INTO SPECIES_ABUNDANCES (ID,species_id, speciation_
↪rate, "
                        "no_individuals,generation) VALUES (?,?,?,?,?);";
        sqlite3_prepare_v2(database, table_command.c_str(), strlen(table_command.c_
↪str()), &stmt, NULL);

        // Start the transaction
        sqlite3_exec(database, "BEGIN TRANSACTION;", NULL, NULL, NULL);
        for(unsigned int i = 0; i < rOut.size(); i++)
        {
            // only do all the export itself if the value of i is not 0
            // if(rOut[i] != 0)
            //{
            string id_var = boost::lexical_cast<std::string>(dSpecRate) + "-" +
                            to_string((unsigned long long)i) + "_" + to_string((long␣
↪double)generation);
            // fixed precision problem - lexical cast allows for printing of very␣
↪small doubles.
            sqlite3_bind_text(stmt, 1, id_var.c_str(), -1, SQLITE_STATIC);
            sqlite3_bind_int(stmt, 2, i);
            sqlite3_bind_double(stmt, 3, dSpecRate);
            sqlite3_bind_int(stmt, 4, rOut[i]);
            sqlite3_bind_double(stmt, 5, generation);
            int step = sqlite3_step(stmt);
            // makes sure the while loop doesn't go forever.
```

```
            time_t start_check, end_check;
            time(&start_check);
            time(&end_check);
            while(step != SQLITE_DONE && (end_check - start_check) < 1)
            {
                step = sqlite3_step(stmt);
                time(&end_check);
            }
            if(step != SQLITE_DONE)
            {
//              cout << endl << rOut[i] << endl;
//              cout << generation << endl;
//              cout << i << endl;
//              cout << dSpecRate << endl;
//              cout << id_var << endl;
                cerr << "SQLITE error code: " << step << endl;
                cerr << "ERROR_SQL_004: Could not insert into database. Check␣
↪destination file has not "
                        "been moved or deleted and that an entry doesn't already␣
↪exist with the same ID."
                     << endl;
                sqlite3_clear_bindings(stmt);
                sqlite3_reset(stmt);
                break;
            }
            sqlite3_clear_bindings(stmt);
            sqlite3_reset(stmt);
            //}
        }

        // execute the command and close the connection to the database
        int rc1 = sqlite3_exec(database, "END TRANSACTION;", NULL, NULL, NULL);
        // Need to finalise the statement
        int rc2 = sqlite3_finalize(stmt);
        if(rc1 != SQLITE_OK || rc2 != SQLITE_OK)
        {
            cerr << "ERROR_SQL_013: Could not complete SQL transaction. Check memory␣
↪database assignment and "
                    "SQL commands. Ensure SQL statements are properly cleared and␣
↪that you are not attempting "
                    "to insert repeat IDs into the database."
                 << endl;
        }
        else
        {
            cout << "\rGenerating new SQL table for speciation rate " << s << "...
↪done!" << endl;
        }
    }
}

void Treelist::createFragmentDatabase(const Fragment& f)
{
    //      cout << "Generating new SQL table for speciation rate " << s << "..." <<␣
↪flush;
    string table_command = "CREATE TABLE IF NOT EXISTS FRAGMENT_ABUNDANCES (ID TEXT␣
↪PRIMARY KEY NOT NULL, fragment "
                           "TEXT NOT NULL, area DOUBLE NOT NULL, size INT NOT NULL, ␣
↪species_id INT NOT NULL, "
```

```
                            "speciation_rate DOUBLE NOT NULL, no_individuals INT NOT␣
↪NULL,generation DOUBLE NOT NULL);";
    sqlite3_exec(database, table_command.c_str(), NULL, NULL, NULL);

    sqlite3_stmt* stmt;
    table_command = "INSERT INTO FRAGMENT_ABUNDANCES (ID, fragment, area, size,␣
↪species_id, speciation_rate, "
                    "no_individuals,generation) VALUES (?,?,?,?,?,?,?,?);";
    sqlite3_prepare_v2(database, table_command.c_str(), strlen(table_command.c_str()),
↪ &stmt, NULL);

    // Start the transaction
    sqlite3_exec(database, "BEGIN TRANSACTION;", NULL, NULL, NULL);
    for(unsigned int i = 0; i < rOut.size(); i++)
    {
        if(rOut[i] != 0)
        {
            string id_var = boost::lexical_cast<std::string>(dSpecRate) + "_" + f.
↪name + "_" +
                            to_string((unsigned long long)i) + "_" + to_string((long␣
↪double)generation);
            // fixed precision problem - lexical cast allows for printing of very␣
↪small doubles.
            sqlite3_bind_text(stmt, 1, id_var.c_str(), -1, SQLITE_STATIC);
            sqlite3_bind_text(stmt, 2, f.name.c_str(), -1, SQLITE_STATIC);
            sqlite3_bind_double(stmt, 3, f.area);
            sqlite3_bind_int(stmt, 4, f.num);
            sqlite3_bind_int(stmt, 5, i);
            sqlite3_bind_double(stmt, 6, dSpecRate);
            sqlite3_bind_int(stmt, 7, rOut[i]);
            sqlite3_bind_double(stmt, 8, generation);
            int step = sqlite3_step(stmt);
            // makes sure the while loop doesn't go forever.
            time_t start_check, end_check;
            time(&start_check);
            time(&end_check);
            while(step != SQLITE_DONE && (end_check - start_check) < 10)
            {
                step = sqlite3_step(stmt);
                time(&end_check);
            }
            if(step != SQLITE_DONE)
            {
                cerr << "SQLITE error code: " << step << endl;
                cerr << "ERROR_SQL_004: Could not insert into database. Check␣
↪destination file has not "
                        "been moved or deleted and that an entry doesn't already␣
↪exist with the same ID."
                     << endl;
                sqlite3_clear_bindings(stmt);
                sqlite3_reset(stmt);
                break;
            }
            sqlite3_clear_bindings(stmt);
            sqlite3_reset(stmt);
        }
    }
    // execute the command and close the connection to the database
```

```
    int rc1 = sqlite3_exec(database, "END TRANSACTION;", NULL, NULL, NULL);
    // Need to finalise the statement
    int rc2 = sqlite3_finalize(stmt);
    if(rc1 != SQLITE_OK || rc2 != SQLITE_OK)
    {
        cerr << "ERROR_SQL_013: Could not complete SQL transaction. Check memory␣
↪database assignment and SQL "
                "commands. Ensure SQL statements are properly cleared and that you␣
↪are not attempting to insert "
                "repeat IDs into the database."
             << endl;
    }
    else
    {
        //          cout << "\rGenerating new SQL table for speciation rate " << s <<
↪"...done!" << endl;
    }
}

void Treelist::exportDatabase(string outputfile)
{
    if(bMem)
    {
        cout << "Writing out to " << outputfile << "..." << flush;
        // Now write the database to the file object.
        sqlite3* outdatabase;

        int rc = sqlite3_open_v2(outputfile.c_str(), &outdatabase, SQLITE_OPEN_
↪READWRITE, "unix-dotfile");
        // check that the connection to file has opened correctly
        if(rc != SQLITE_OK && rc != SQLITE_DONE)
        {
            // attempt other output method
            sqlite3_close(outdatabase);
            rc = sqlite3_open(outputfile.c_str(), &outdatabase);
            if(rc != SQLITE_OK && rc != SQLITE_DONE)
            {
                cerr << "ERROR_SQL_016: Connection to output database cannot be␣
↪opened. Check write access "
                        "on output folder. Error code: "
                     << rc << "." << endl;
            }
        }

        // create the backup object to write data to the file from memory.

        sqlite3_backup* backupdb;
        backupdb = sqlite3_backup_init(outdatabase, "main", database, "main");
        if(!backupdb)
        {
            cerr << "ERROR_SQL_003: Could not backup to SQL database. Check␣
↪destination file has not been "
                    "moved or deleted."
                 << endl;
        }
        // Perform the backup
        rc = sqlite3_backup_step(backupdb, -1);
        if(rc != SQLITE_OK && rc != SQLITE_DONE)
```

```
        {
            cerr << "ERROR_SQL_016: Connection to output database cannot be opened.␣
↪Check write access on "
                    "output folder. Error code: "
                << rc << "." << endl;
        }
        rc = sqlite3_backup_finish(backupdb);
        if(rc != SQLITE_OK && rc != SQLITE_DONE)
        {
            cerr << "ERROR_SQL_016: Connection to output database cannot be opened.␣
↪Check write access on "
                    "output folder. Error code: "
                << rc << "." << endl;
        }
        //          cout << "rc: " << rc << endl;
        sqlite3_close(outdatabase);
        sqlite3_close(database);
        cout << "done!" << endl;
    }
    else
    {
        cout << "Closing file..." << flush;
        sqlite3_close(database);
        cout << "\rClosing file...done!" << endl;
    }
    //      database
}

void Treelist::recordSpatial()
{
    cout << "Recording spatial data for speciation rate " << dSpecRate << "..." <<␣
↪flush;
    string table_command = "CREATE TABLE IF NOT EXISTS SPECIES_LOCATIONS (ID TEXT␣
↪PRIMARY KEY NOT NULL, species_id INT "
                           "NOT NULL, speciation_rate DOUBLE NOT NULL, x INT NOT NULL,
↪ y INT NOT NULL,generation "
                           "DOUBLE NOT NULL);";
    sqlite3_exec(database, table_command.c_str(), NULL, NULL, NULL);
    sqlite3_stmt* stmt;
    table_command =
        "INSERT INTO SPECIES_LOCATIONS (ID,species_id, speciation_rate, x,y,
↪generation) VALUES (?,?,?,?,?,?);";
    sqlite3_prepare_v2(database, table_command.c_str(), strlen(table_command.c_str()),
↪ &stmt, NULL);
    //      cout << "test1" << endl;
    // Start the transaction
    sqlite3_exec(database, "BEGIN TRANSACTION;", NULL, NULL, NULL);
    // Make sure only the tips which we want to check are recorded
    //      cout << "nodes.size(): " << nodes.size() << endl;
    for(unsigned int i = 0; i < nodes.size(); i++)
    {
        //          cout << nodes[i].getExistance() << endl;
        if(samplemask.getVal(nodes[i].getXpos(), nodes[i].getYpos()) && nodes[i].
↪isTip() &&
           nodes[i].getExistance() && doubleCompare(nodes[i].getGeneration(),␣
↪generation, 0.0001))
        {
            string id_var = boost::lexical_cast<std::string>(dSpecRate) + "-" +
```

```
                            to_string((unsigned long long)i) + "_" + to_string((long
↪double)generation);
            //              cout << id_var << endl;
            // fixed precision problem - lexical cast allows for printing of very
↪small doubles.
            sqlite3_bind_text(stmt, 1, id_var.c_str(), -1, SQLITE_STATIC);
            sqlite3_bind_int(stmt, 2, nodes[i].getSpeciesID());
            sqlite3_bind_double(stmt, 3, dSpecRate);
            sqlite3_bind_int(stmt, 4, nodes[i].getXpos());
            sqlite3_bind_int(stmt, 5, nodes[i].getYpos());
            sqlite3_bind_double(stmt, 6, nodes[i].getGeneration());
            int step = sqlite3_step(stmt);
            // makes sure the while loop doesn't go forever.
            time_t start_check, end_check;
            time(&start_check);
            time(&end_check);
            while(step != SQLITE_DONE && (end_check - start_check) < 10 && step !=
↪SQLITE_OK)
            {
                step = sqlite3_step(stmt);
                time(&end_check);
            }
            if(step != SQLITE_DONE)
            {
                cerr << "SQLITE error code: " << step << endl;
//              cerr << nodes[i].getSpeciesID() << endl;
//              cerr << nodes[i].getXpos() << endl;
//              cerr << id_var << endl;
                cerr << "ERROR_SQL_004: Could not insert into database. Check
↪destination file has not "
                        "been moved or deleted and that an entry doesn't already
↪exist with the same ID."
                     << endl;
                break;
            }
            sqlite3_clear_bindings(stmt);
            sqlite3_reset(stmt);
        }
    }
    //      cout << "test2" << endl;
    // execute the command and close the connection to the database
    int rc1 = sqlite3_exec(database, "END TRANSACTION;", NULL, NULL, NULL);
    // Need to finalise the statement
    int rc2 = sqlite3_finalize(stmt);
    if(rc1 != SQLITE_OK || rc2 != SQLITE_OK)
    {
        cerr << "ERROR_SQL_013: Could not complete SQL transaction. Check memory
↪database assignment and SQL "
                "commands. Ensure SQL statements are properly cleared and that you
↪are not attempting to insert "
                "repeat IDs into the database."
             << endl;
    }
    else
    {
        cout << "\rRecording spatial data for speciation rate " << dSpecRate << "...
↪done!" << endl;
    }
```

```
}

void Treelist::calcFragments(string fragment_file)
{
//  for(unsigned int i = 0; i < nodes.size(); i++)
//  {
//      if(nodes[i].isTip() && !samplemask.getVal(nodes[i].getXpos(), nodes[i].
↪getYpos()))
//      {
            //              cerr << "ERROR (x,y): " << nodes[i].getXpos() << ", " <
↪<nodes[i].getYpos() <<
            //endl;
//      }
//  }
    // Loop over every grid cell in the samplemask to determine if it is the start␣
↪(top left corner) of a fragment.
    // Note that fragment detection only works for squares and rectangles. Adjacent␣
↪squares and rectangles will be
    // treated as separate fragments if they are different sizes.
    // Downwards shapes are prioritised (i.e. a vertical rectangle on top of a␣
↪horizontal rectangle will produce 3
    // fragments instead of two - this is a known bug).
    if(fragment_file == "null")
    {
        unsigned int fragment_number = 0;
        for(int i = 0; i < samplemask.GetCols(); i++)
        {
            for(int j = 0; j < samplemask.GetRows(); j++)
            {
                bool in_fragment = false;
                // Make sure is isn't on the top or left edge
                if(samplemask.getVal(i, j))
                {
                    if(i > 0 && j > 0)
                    {
                        // Perform the check
                        if(samplemask.getVal(i - 1, j) || samplemask.getVal(i, j - 1))
                        {
                            // then it is not a fragment
                            in_fragment = false;
                        }
                        else
                        {
                            in_fragment = true;
                        }
                    }
                    // if it is on an edge, we need to check the fragment
                    else
                    {
                        // if it is on the left edge we need to check above it - if␣
↪there is forest
                        // there, it is not a fragment.
                        if(i == 0 && j > 0)
                        {
                            if(!samplemask.getVal(i, j - 1))
                            {
                                in_fragment = true;
                            }
```

```
                        }
                        // if it is on the top edge, need to check to the left of it -
↪  if there is
                        // forest there, it is not a fragment.
                        else if(j == 0 && i > 0)
                        {
                            if(!samplemask.getVal(i - 1, j))
                            {
                                in_fragment = true;
                            }
                        }
                        else if(i == 0 && j == 0)
                        {
                            in_fragment = true;
                        }
                    }
                }
                if(in_fragment)
                {
                    // Now move along the x and y axis (separately) until we hit a␣
↪non-forest patch.
                    // This marks the edge of the fragment and the value is recorded.
                    bool x_continue = true;
                    bool y_continue = true;
                    unsigned long x, y;
                    x = i;
                    y = j;
                    fragment_number++;
                    // Also need to check that fragments that lie partly next to each␣
↪other aren't
                    // counted twice.
                    // So count along the x axis until we hit non-habitat. Then count␣
↪down the y axis
                    // checking both extremes of the square for non-habitat.
                    // Perform a check on the x axis to make sure that the square␣
↪above is empty, as
                    // fragments give priority in a downwards motion.
                    while(x_continue)
                    {
                        x++;
                        if(samplemask.getVal(x, j))
                        {
                            // Check we're not on top edge of the map.
                            if(j > 0)
                            {
                                // if the cell above is non-fragment then we don't␣
↪need to
                                // continue (downwards fragments get priority).
                                if(samplemask.getVal(x, j - 1))
                                {
                                    x_continue = true;
                                }
                                else
                                {
                                    x_continue = false;
                                }
                            }
                            else
```

```
                            {
                                x_continue = true;
                            }
                        }
                        else
                        {
                            x_continue = false;
                        }
                    }
                    while(y_continue)
                    {
                        y++;
                        // Make sure both extremes of the rectangle are still within␣
↪patch.
                        if(samplemask.getVal(i, y) && samplemask.getVal(x - 1, y))
                        {
                            y_continue = true;
                        }
                        else
                        {
                            y_continue = false;
                        }
                    }
                    // Create the fragment to add.
                    Fragment to_add;
                    to_add.name = to_string((long long)fragment_number);
                    to_add.x_west = i;
                    to_add.x_east = x - 1;
                    to_add.y_north = j;
                    to_add.y_south = y - 1;
                    // calculate the square area of the plot and record it.
                    to_add.area = (x - i) * (y - j);
                    // Now store the size of the fragment in the vector.
                    fragments.push_back(to_add);
                }
            }
        }
    }
    else
    {
#ifdef use_csv
        cout << "Importing fragments from " << fragment_file << endl;
        // There is a config file to import - here we use a specific piece of import␣
↪code to parse the csv file.
        // first count the number of lines
        int number_of_lines = 0;
        string line;
        ifstream fragment_configs(fragment_file);
        while(getline(fragment_configs, line))
        {
            number_of_lines++;
        }
        //          cout << "Number of lines in text file: " << number_of_lines <<␣
↪endl;
        fragment_configs.close();
        io::LineReader in(fragment_file);
        // Keep track of whether we've printed to terminal or not.
        bool bPrint = false;
```

```
        fragments.resize(number_of_lines);
//      cout << "size: "  << fragments.capacity() << endl;
        for(int i = 0; i < number_of_lines; i++)
        {

            //              cout << i << endl;
            char* line = in.next_line();
//          cout << line << endl;
            if(line == nullptr)
            {
                if(!bPrint)
                {
                    cerr << "Input dimensions incorrect - read past end of file." <<
↪endl;
                    bPrint = true;
                }
                break;
            }
            else
            {
                char* dToken;
                dToken = strtok(line, ",");
                for(int j = 0; j < 6; j++)
                {
                    //                      cout << j << endl;
                    if(dToken == nullptr)
                    {
                        if(!bPrint)
                        {
                            cerr << "Input dimensions incorrect - read past end of
↪file."
                                 << endl;
                            bPrint = true;
                        }
                        break;
                    }
                    else
                    {
                        //                          cout << "-" << endl;
                        switch(j)
                        {
                            case 0:
                            fragments[i].name = string(dToken);
                            break;
                            case 1:
                            fragments[i].x_west = atoi(dToken);
                            break;
                            case 2:
                            fragments[i].y_north = atoi(dToken);
                            break;
                            case 3:
                            fragments[i].x_east = atoi(dToken);
                            break;
                            case 4:
                            fragments[i].y_south = atoi(dToken);
                            break;
                            case 5:
                            fragments[i].area = atof(dToken);
```

```
                            break;
                        }
                        dToken = strtok(NULL, ",");
                    }
                }
            }
        }
#endif
#ifndef use_csv
        cerr << "Cannot import fragments from " << fragment_file << " without fast-
↪cpp-csv-parser." << endl;
        cerr << "Make sure the program has been compiled with -D use_csv." << endl;
#endif
    }
    // for(unsigned int i =0; i < fragments.size(); i++ )
    //{
    // check samples match
    // if(!samplemask.getVal(fragments[i].x_west,fragments[i].y_north))
    //{
    // cout << fragments[i].name << " " << fragments[i].x_west << " " << fragments[i].
↪y_north << " " <<
    // fragments[i].x_east << " " << fragments[i].y_south << " " << fragments[i].area
↪<< endl;
    //}
    //}
    cout << "Completed fragmentation analysis: " << fragments.size() << " fragments␣
↪identified." << endl;
}

void Treelist::applyFragments()
{
    //      calculateFragments(); // don't need to do this each time - just once per␣
↪simulation.
    // For each fragment in the vector, perform the analysis and record the data in␣
↪to a new data object, which will
    // then be outputted to an SQL file.
    for(unsigned int i = 0; i < fragments.size(); i++)
    {
        //          cout  << fragments[i].x_west << endl;
        cout << "\rApplying fragments... " << (i + 1) << "/" << fragments.size() << "␣
↪      " << flush;
        // Set the new samplemask to the fragment
        samplemask.setFragment(fragments[i]);
        // Now filter only those lineages which exist in the fragments.
        // We also want to count the number of individuals that actually exist
        unsigned long iSpecCount = 0;
        for(unsigned long j = 0; j < nodes.size(); j++)
        {
            if(nodes[j].isTip() && samplemask.getVal(nodes[j].getXpos(), nodes[j].
↪getYpos()) &&
               doubleCompare(nodes[j].getGeneration(), generation, 0.0001))
            {
                // if they exist exactly in the generation of interest.
                nodes[j].setExistance(true);
                iSpecCount++;
            }
            else if(nodes[j].isTip())
            {
```

```
                nodes[j].setExistance(false);
            }
        }
        fragments[i].num = iSpecCount;
        /* NO LONGER REQUIRED DUE TO IMPLEMENTATION CHANGE
        vector<unsigned long> all_species;
        all_species.reserve(iSpecCount);
        for(unsigned int i=0;i<=nodes.size();i++)
        {
                if(nodes[i].isTip() && nodes[i].getExistance())//&&nodes[i].
↪getExistance()) // count all speciation
        events, not just the ones that exist!
                {
                        all_species.push_back(nodes[i].getSpeciesID());
                }
                if(nodes[i].getExistance()&&nodes[i].getSpeciesID()==0)
                {
                        cerr << "ERROR_SQL_006: Species exists but hasn't been␣
↪assigned a species ID. Check the tree
        speciation calculations." << endl;
                }
        }
        // sory the vector
        sort(all_species.begin(),all_species.end());
        auto last = unique(all_species.begin(),all_species.end());
        all_species.erase(last,all_species.end());
        iSpecies = all_species.size();
         * */
        // Now calculate the species abundance. This will create a vector with lots␣
↪of zeros in it. However, the
        // database creation will filter these out.
        calcSpeciesAbundance();
        createFragmentDatabase(fragments[i]);
        //          cout << "done!" << endl;
    }
    samplemask.removeFragment();
    cout << "\rApplying fragments... "
         << "done!      " << endl;
}

bool checkSpeciation(long double random_number, long double speciation_rate, int␣
↪number_of_generations)
{
    // bool result1, result2, result3, result4;
    long double res = double(1.0 - pow(double(1.0 - speciation_rate), double(number_
↪of_generations)));
    if(random_number <= res)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}
```

Includes

- `Treelist.h` (*File Treelist.h*)

Functions

- *Function checkSpeciation*

File Treelist.h

Definition (`Treelist.h`)

Program Listing for File Treelist.h

- Return to documentation for *File Treelist.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for␣
↪full license details.
// This code is a used to analyse a list of Treenode objects to generate species␣
↪abundances for a given speciation rate.
// For use within Coal_sim v3.1+
// Author: Samuel Thompson
// Contact: samuel.thompson14@imperial.ac.uk or thompsonsed@gmail.com
// Version 1.1
//#include "Treenode.h"
#ifndef TREELIST
#define TREELIST
#include "Treenode.h"
#include "Matrix.h"
#include <math.h>
#include <sqlite3.h>
#include <cstring>
#include <cmath>
#include <stdexcept>
#include <string>
# include <boost/filesystem.hpp>
#include <boost/lexical_cast.hpp>
 #include "CustomExceptions.h"

using namespace std;
using std::string;

bool checkSpeciation(long double random_number, long double speciation_rate, int␣
↪number_of_generations)
;

// A class containing the fragment limits as x,y coordinates.
struct Fragment
{
    // the name for the fragment (for reference purposes)
    string name;
    // coordinates for the extremes of the site
    unsigned long x_east, x_west, y_north, y_south;
```

```
    // the number of lineages in the fragment.
    unsigned long num;
    double area;
};

// Class for creating the sample matrix object for easy referencing

class Samplematrix :  public Matrix<bool>
{
private:
    bool bIsNull;
    bool bIsFragment;
    Fragment fragment;
public:
    Samplematrix()
    {
        bIsFragment = false;
        bIsNull = false;
    }

    void setIsNull(bool b)
    {
        bIsNull = b;
    }

    bool getIsNull()
    {
        return bIsNull;
    }

    bool getTestVal(unsigned int xval, unsigned int yval)
    {
        return matrix[yval][xval];
    }

    bool getVal(unsigned int xval, unsigned int yval)
    {
        if(bIsFragment)
        {
            if(fragment.x_west <= xval  && xval <= fragment.x_east && fragment.y_
↪north <= yval  && yval <= fragment.y_south)
            {
//              cout << "FRAGMENT TRUE" << endl;
//              cout << "x,y" << xval << ", " << yval << "frag: w,e,n,s: " <<␣
↪fragment.x_west << ", " << fragment.x_east << ", " << fragment.y_north << ", " <<␣
↪fragment.y_south << endl;
                return true;
            }
            else
            {
//              cout << "FRAGMENT FALSE" << endl;
                return false;
            }
        }
        if(bIsNull)
        {
            return true;
        }
```

```
        else
        {
            return matrix[yval][xval];
        }
    }

     void setFragment(Fragment fragment_in)
     {
            fragment = fragment_in;
//          cout << "W,E,N,S: " << fragment.x_west << ", " << fragment.x_east << ", "
↪<< fragment.y_north << ", " << fragment.y_south << endl;
            bIsFragment = true;
     }

     void removeFragment()
     {
         bIsFragment = false;
     }
};




class Treelist
{
private:
    bool bMem; // boolean for whether the database is in memory or not.
    sqlite3 * database;
    Row<Treenode> &nodes; // in older versions this was called list. Changed to avoid␣
↪confusion with the built-in class.
    Row<unsigned int> rOut;
    double dSpecRate;
    unsigned long iSpecies;
    long double generation; // the time of interest for the simulation
    bool bSample; // checks whether the samplemask has already been imported.
    string samplemaskfile; // stores the name of the file object for referencing.
    Samplematrix samplemask; // the samplemask object for defining the areas we want␣
↪to sample from.
    vector<Fragment> fragments; // a vector of fragments for storing each fragment's␣
↪coordinates.
    // the minimum speciation rate the original simulation was run with (this is read␣
↪from the database SIMULATION_PARAMETERS table)
    double min_spec_rate;
    // The dimensions of the sample grid size.
    unsigned long grid_x_size;
    unsigned long grid_y_size;

public:
    Treelist(Row<Treenode>&r):nodes(r)
    {
        bMem = false;
        dSpecRate =0;
        iSpecies =0;
        bSample = false;
        generation = 0;
    }
```

```
    void setList(Row<Treenode> &l);

    void setDatabase(sqlite3 * dbin);

    bool hasSetMem();

    double getMinimumSpeciation();

    void importSamplemask(string sSamplemask);

    unsigned long countSpecies();

    bool doubleCompare(double d1, double d2, double epsilon);

    unsigned long calcSpecies(double s);

    void calcSpeciesAbundance();

    void resetTree();

    void detectDimensions(string db);


    void importData(string inputfile);

    void setGeneration(long double generationin)
    ;

    void createDatabase(double s);

    void createFragmentDatabase(const Fragment &f);

    void exportDatabase(string outputfile);

    void recordSpatial();

    void calcFragments(string fragment_file);

    void applyFragments();
};

#endif
```

Includes

- `CustomExceptions.h` (*File CustomExceptions.h*)
- `Matrix.h` (*File Matrix.h*)
- `Treenode.h` (*File Treenode.h*)
- `boost/filesystem.hpp`
- `boost/lexical_cast.hpp`
- `cmath`
- `cstring`

- `math.h`
- `sqlite3.h`
- `stdexcept`
- `string`

Included By

- *File Tree.h*
- *File SpeciationCounter.cpp*
- *File Treelist.cpp*

Classes

- *Struct Fragment*
- *Class Samplematrix*
- *Class Treelist*

Functions

- *Function checkSpeciation*

File Treenode.h

Definition (`Treenode.h`)

Program Listing for File Treenode.h

- Return to documentation for *File Treenode.h*

```
//This file is part of NECSim project which is released under BSD-3 license.
//See file **LICENSE.txt** or visit https://opensource.org/licenses/BSD-3-Clause) for
↪full license details.
// Datapoint.cpp version 1.01
// Author - Samuel Thompson - Imperial College London
// with large use of code supplied by James Rosindell (Imperial College London)
// This file contains the datapoint class for usage in coalescence simulations.
#ifndef TREENODE
#define TREENODE
# define version1_01

/*************************************************************
                        TREENODE OBJECT
 *************************************************************/
#include <stdio.h>
#include <iostream>
using namespace std;
class Treenode
```

```
{

private:
    bool tip;
    // 0 means that this node is just here to mark a coalescense
    // and therefore this node of no real other relevance
    // 1 means that this node is a leaf node and counts towards diversity
    unsigned long parent;
    // this stores the parent of the individual
    // 0 means there is no parent - we are at the end of the tree
    // (as far as has been calculated)
    bool speciated;
    // true if this lineage has speciated in which case it should not have a parent
    // because under the present implementation lineages are not traced beyond␣
↪speciation

    // boolean for checking whether the lineage actually exists at the end. If all␣
↪children of the lineages have speciated, then the lineage no longer exists.
    bool does_exist;

    // the species identity of the node
    unsigned long species_id;

    // the following 4 variables describe the position of the lineage in the present␣
↪day.
    unsigned long xpos;
    // x position
    unsigned long ypos;
    // y position
    long xwrap;
    // number of wraps of x around the torus
    long ywrap;
    // number of wraps of y around the torus

    long double dSpec;
    // the speciation probability. This needs to be multiplied by the number of␣
↪generations in order to generate the actual probability.
    unsigned long iGen;
    long double generation_added;
public:
    Treenode() : tip(false),parent(0),speciated(false),does_exist(false),species_
↪id(0),xpos(0),ypos(0),xwrap(0),ywrap(0), dSpec(0), iGen(0),generation_added(0)
    {

    }

    ~Treenode()
    {

    }

    void setup(bool z , unsigned long xp , unsigned long yp , long xi , long yi)
    {
        tip = z;
        parent = 0;
        speciated = false;

        species_id = 0;
```

```
        xpos = xp;
        ypos = yp;
        xwrap = xi;
        ywrap = yi;
        dSpec =0;
        iGen = 0;
        generation_added = 0;
    }

    void setup(bool z , unsigned long xp , unsigned long yp , long xi , long yi, long␣
↪double generation)
    {
        tip = z;
        parent = 0;
        speciated = false;

        species_id = 0;

        xpos = xp;
        ypos = yp;
        xwrap = xi;
        ywrap = yi;
        dSpec =0;
        iGen = 0;
        generation_added = generation;
    }
    // standard setters

    void setExistance(bool b)
    {
        does_exist = b;
    }

    void setParent(unsigned long x)
    {
        parent = x;
    }

    void qReset()
    {
        species_id = 0;
        does_exist = false;
        speciated = false;
    }

    void setPosition(long x,long y, long xw, long yw)
    {
        xpos = x;
        ypos = y;
        xwrap = xw;
        ywrap = yw;
    }

    void setSpec(long double d)
    {
        dSpec = d;
    }
```

```
    void setIGen(unsigned long g)
    {
        iGen = g;
    }

    void setGeneration(long double d)
    {
        generation_added = d;
    }

    void setSpeciation(bool s)
    {
        speciated = s;
    }

    void burnSpecies(unsigned long idin)
    {
        if (species_id == 0)
        {
            species_id = idin;
        }
    }

    void setTip(bool b)
    {
        tip = b;
    }

    void resetSpecies()
    {
        species_id =0;
    }

    void increaseGen()
    {
        iGen++;
    }
    // we don't allow the other variables to be changed
    // because they only need to be set once at the start of the coalescence
    // it's actually safer to leave out setters.
    // similarly we don't allow speciation to be changed once it has been set.

    // standard getters

    bool getExistance()
    {
        return does_exist;
    }

    bool isTip()
    {
        return tip;
    }

    unsigned long getParent()
    {
        return parent;
```

```
    }

    unsigned long getXpos()
    {
        return xpos;
    }

    unsigned long getYpos()
    {
        return ypos;
    }

    long getXwrap()
    {
        return xwrap;
    }

    long getYwrap()
    {
        return ywrap;
    }

    bool hasSpeciated()
    {
        return (speciated);
    }

    unsigned long getSpeciesID()
    {
        return (species_id);
    }

    long double getSpecRate()
    {
        return dSpec;
    }

    unsigned long getGenRate()
    {
        return iGen;
    }

    long double getGeneration()
    {
        return generation_added;
    }
    // routines

    void speciate()
    {
        speciated = true;
//      parent = -1;
    }

    friend ostream& operator<<(ostream& os,const Treenode& t)
    {
        //os << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        //
```

```
        os <<t.tip << "," << t.parent << "," << t.speciated << "," << t.does_exist <<
↪"," << t.species_id << "," << t.xpos << "," << t.ypos << "," << t.xwrap << ",";
        os << t.ywrap << "," << t.dSpec << "," << t.iGen << "," << t.generation_added
↪<< ",";
        return os;
    }

    friend istream& operator>>(istream& is,Treenode& t)
    {
        //is << m.numRows<<" , "<<m.numCols<<" , "<<endl;
        char delim;
        is >>t.tip >> delim >> t.parent >> delim >> t.speciated >> delim >> t.does_
↪exist >> delim >> t.species_id >> delim >> t.xpos >> delim;
        is >> t.ypos >> delim >> t.xwrap >> delim >> t.ywrap >> delim >> t.dSpec >>␣
↪delim >> t.iGen >> delim >> t.generation_added >> delim;
        return is;
    }
};

#endif
```

Includes

- iostream
- stdio.h

Included By

- *File Tree.h*
- *File Treelist.h*

Classes

- *Class Treenode*

Defines

- *Define version1_01*

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

EIGHT

CITATIONS

BIBLIOGRAPHY

[Hubbell2001] Hubbell, S. P. (2001). The Unified Neutral Theory of Biodiversity and Biogeography. Monographs in Population Biology, 17(3), 375. http://doi.org/10.1016/S0006-3207(02)00228-8

PYTHON MODULE INDEX

p