e3SIM
Epidemiological-ecological-evolutionary simulation framework for genetic epidemiology

Perry Xu, Shenni Liang, Jae Hahn, Vivian Zhao, Wai Tung 'Jack' Luo, Ben Haller

Dept. of Computational Biology
Cornell University, Ithaca, NY 14853

Correspondence: placeholder@gmail.com
Update on July 11, 2024

Author Contributions:

Peiyu Xu
Shenni Liang
Jae Hahn
Vivian Zhao
Wai Tung 'Jack' Luo
Ben Haller
Jaehee Kim

Acknowledgements:

Wai Tung 'Jack' Lo, Andrew Clark, TB cohort.

Citation:

To cite e3SIM in a publication, please cite:
placeholder for citation lines

URL:

placeholder.com

License:

Disclaimer:

Contents

Glossary

drug resistance	The trait affecting the probability of pathogen survival during host treatment. 25, 37, 43
effective contact	An effective contact involves a pair of hosts connected by an edge, where one host is infected (the infector) and the other is capable of being infected (the infectee). 37
epoch	A range of consecutive ticks. Within each epoch, the base rates for compartmental transitions and the genetic architecture for each trait remain constant.. 36, 37, 41
host	The larger organism that harbours one or more pathogens. 7, 36
initial sequences	The pathogen genome(s) that infect a fully susceptible host population as outbreak simulation starts. 17
pathogen	The organism whose movements and genomes are being modeled. 7
tick	Analogous to the definition of **tick** defined in population genetics simulation framework SLiM. It is the time unit of the epidemiological events, the time unit of mutation happening, and the unit of branch length of the genealogy output. 18, 24, 34, 36, 41
transmissibility	The trait affecting the probability of pathogen transmission per contact. 25, 37, 43

Glossary

Preface

Contents

0.1 An Overview of e3SIM

e3SIM (**E**pidemiological-**e**cological-**e**volutionary **sim**ulation framework for genetic epidemiology) is an agent-based, discrete, forward-in-time outbreak simulator that models pathogen evolution and transmission dynamics featuring host population contact structures. The tool consists of 3 components: pre-simulation, main simulation, and post-simulation, all accessible through the command line. Additionally, we offer a graphic user interface (GUI) for conducting all the pre-simulation modules and setting a configuration for the main module. The GUI is structured to guide users step-by-step through the process of configuring and initiating simulations, providing clear and intuitive interaction with the software's features.

0.1.1 Introduction

e3SIM implements an agent-based, discrete and forward-in-time approach to simultaneously simulate the transmission dynamics and molecular evolution of a haploid transmissible pathogen population within a static host population structure. It highlights the coupling of evolutionary and epidemiological processes and explicitly models transmission on top of a host contact network. Utilizing a SEIRS compartmental model, e3SIM adeptly balances the simplification of real-world processes with the theoretical constructs inherent in both epidemiological and quantitative genetic models. For a thorough introduction of the background and principle which e3SIM is built upon, please refer to our manuscript.

The package integrates Python, R, and SLiM scripts. Its first component, the pre-simulation modules generates essential input files for the main module to run. The second part executes the main simulation using SLiM — a script-based simulator designed for population genetics — as the back-end engine. We allow various permissible transitions between compartments, where transmissions and treatments are affected by genetic-based trait values. The third component, executed together with the main module, includes analysis and visualizations for the simulation output, including raw whole genome sequence, summary statistics, and graphic representations of results, such as the phylogeny of the transmission tree and the compartments' trajectories averaged over all simulation replicates.

0.1.2 A quick summary of usage

e3SIM contains 5 modules. 4 modules belong to the pre-simulation component: *NetworkGenerator* (Section 2.1), *SeedGenerator* (Section 2.2), *GeneticEffectGenerator* (Section 2.3), *HostSeedMatcher* (Section 2.4), and

1 module belongs to the main component: *OutbreakSimulator* (Chapter 3). The software can be run in one of the following ways.

- **Sequential**: Run each module individually and sequentially on the command line.
- **Streamlined**: Run programs streamlined on the command line with a complete configuration file. In this option, users can run all pre-simulation, main, and post-processing modules together with a single command by providing a comprehensive configuration file.
- **GUI**: Run pre-simulation modules sequentially in GUI and then execute *OutbreakSimulator* on the command line.

The pre-simulation modules generate several specific input files necessary for the main simulation. Users can also create these input files themselves, but the files should follow specific format. The required files for the main simulation include:

1. **Host contact network**: a tab-delimited adjacency list file defining the host contact network, where each row represents a host in order and lists its direct connections. This file is generated by the `NetworkGenerator` module (Section 2.1).
2. **Seeding pathogen sequences**: One file for each seed in the Variant Call Format (`VCF`) format containing the mutation profiles of the seeding pathogen sequences, which are introduced to the host population at the start of the simulation. These files are generated by the `SeedGenerator` module (Section 2.2) through stochastic simulation from the reference genome, unless users choose to seed the host population with the reference genome. The phylogeny of the seeding sequences can also be provided or generated in the Newick (`NWK`) format.
3. **Genetic architecture specification**: A `CSV` file detailing the genetic architecture of pathogen genomes, specifying the effect sizes for traits such as transmissibility and drug resistance associated with mutations in designated genetic elements. This file is generated by the `GeneticEffectGenerator` module (Section 2.3).
4. **Host-seed assignment**: A `csv` file that maps each seeding pathogen sequence to a specific host within the contact network. This file is generated by the `HostSeedMatcher` module (Section 2.4).
5. **Main module configuration**: A `JSON` file that provides the epi-eco-evo parameters used in the main module. If the GUI is used to run the pre-simulation modules, this file will be generated by the GUI. For users who prefer directly interacting with the configuration file, a template of the configuration is provided in our github repository for reference.

Templates for all these necessary input files are available in our Github repository. If the user chooses to create these input files not via the e3SIM pre-simulation modules but by themselves, we still recommend running the relevant pre-simulation modules in the `user_input` mode (Adding `-method user_input` flag to each pre-simulation module). This will validate the user-provided input files and make a copy of the files with fixed file names in the working directory specified.

0.2 Installation

e3SIM is currently available by `git clone` from github, along with necessary environment settings. For a full instruction on installing e3SIM, please follow the steps in README.md. To check for updates and open issues, please refer to our github page e3SIM by Kim Lab.

Part I

Pre-simulation programs

Chapter 1

Overview

Before diving into how to navigate the software, let us take a look at the prerequisites for running the pre-simulation modules:

- A **reference genome** in `FASTA` format
- Path to a **working directory** (All e3SIM pre-simulation modules, when running on the command line, require a consistent path to the working directory (provided by `-wkdir` option), so that all input files will be generated in the same directory.

Other inputs may be required depending on specific user needs. For example, in order to randomly generate a genetic architecture for the genome, a `GFF` (general feature format) file is required when running ***GeneticEffectGenerator*** program in random generation mode. We will introduce all required and optional inputs below and include more details in the respective chapters.

Each pre-simulation module should be run in order to ensure proper functioning of dependent programs. Below is a typical workflow of running pre-simulation modules:

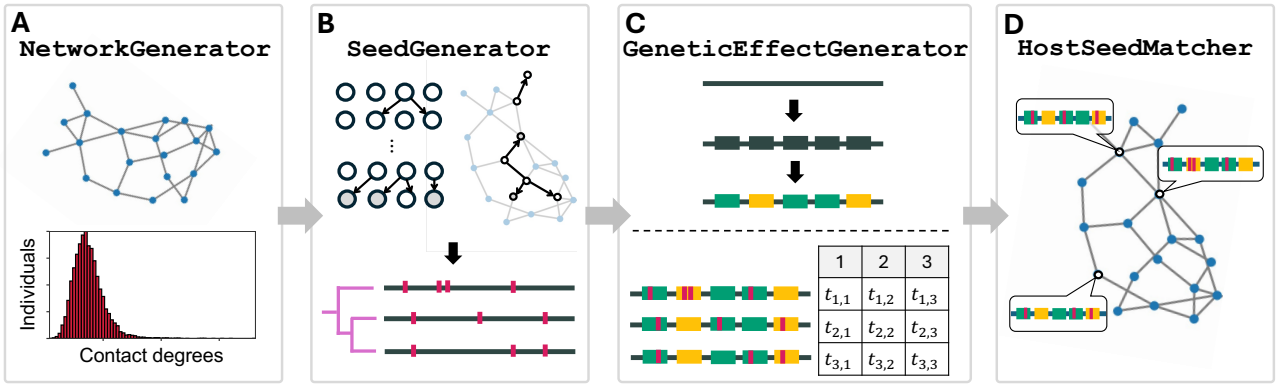


Figure 1.1: **Pre-simulaton modules.**

1. Run ***NetworkGenerator*** to create the host population's contact network. The network is needed for ***OutbreakSimulator***, but also for ***SeedGenerator*** if you want to generate seed sequences using a burn-in by an epidemiological model, for example.
2. Run ***SeedGenerator*** if you want your seeded sequences to be different from the provided reference genome. If you want to use the unchanged reference genome as the initial sequence for all your seeds, this step could be skipped.

3. Run ***GeneticEffectGenerator*** if you want to enable non-neutral mutations in your simulation. When this step is skipped, mutations are still modeled but do not affect the trait values of the pathogens.

4. Run ***SeedHostMatcher*** after ***NetworkGenerator*** and ***SeedGenerator*** to match each initial sequence to a specific host.

In the following chapters, different options to interact with the pre-simulation modules are introduced.

Chapter 2

Run programs one by one

Contents

2.1 NetworkGenerator

e3SIM models the host population and their contact network explicitly. Hosts are modeled as "subpopulations" in SLiM. The host contact network is modeled as an undirected, unweighted graph, with nodes representing hosts and edges indicating contacts between them. The network structure remains static throughout the main simulation. In the main module, only connected hosts can transmit pathogens between each other.

NetworkGenerator is used to generate an adjacency list in the working directory that stores the host contact network information. To run *NetworkGenerator*, the required inputs are working directory (`-wkdir`), host population size (`-popsize`), and method of network input (`-method`). Alternatively, you can provide a custom network instead of using *NetworkGenerator* to generate one, but you can still execute this program to perform a validation check on your network file. Here is how you can run *NetworkGenerator*:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize ${HOST_SIZE} \
    -method ${METHOD} \
    [Other params]
```

Listing 2.1: NetworkGenerator Usage

We introduce all the options:

- `wkdir`: [*REQUIRED] (`string`)
  Absolute path to the working directory, which should be consistent in one workflow for running each module.

- `popsize`: [*REQUIRED] (`integer`)
  Size of the host population, i.e. Number of nodes in the network graph.

- `method`: [*REQUIRED] (`string`) `user_input` or `randomly_generate`
  The method of generating the network, which can only be one of the two methods above. By specifying

`user_input`, users are required to provide your own contact network file by `-path_network`. By specifying `randomly_generate`, users are required to provide a random network model by `-model` and corresponding parameters.

- `path_network`: [*OPTIONAL] (`string`)
  Required if `method` is specified as `user_input`. Absolute path to the customized contact network file. The file has to be in an adjacency list format.

- `model`: [*OPTIONAL] (`string`) `ER` or `BA` or `RP`
  Required if `method` is specified as `randomly_generate`. The random graph model you want to apply for generating the contact network. The choices are `ER` (Erdős–Rényi network), `BA` (Barabási–Albert network) or `RP` (Random partition network).

- `p_ER`: [*OPTIONAL] (`float`)
  Required if `model` is specifed as `ER`. It is the parameter for generating an Erdős–Rényi network, specifying the probability of presence of every possible contact between two host individuals.

- `rp_size`: [*OPTIONAL] (`list of ints`)
  Required if `model` is specifed as `RP`. It is the parameter for generating a random partition network, specifying the size of each partition. As of now, we only support a two-partition model, thus this option has to be an integer list of size 2, which sums up to be the size of the host population provided by `-popsize`.

- `p_within`: [*OPTIONAL] (`list of floats`)
  Required if `model` is specifed as `RP`. It is the parameter for generating a random partition network, specifying the probability of presence of every possible contact between host individuals within the same partition. As of now, we only support a two-partition model, thus this option has to be a float list of size 2, each representing a contact probability between individuals within each partition.

- `p_between`: [*OPTIONAL] (`float`)
  Required if `model` is specifed as `RP`. It is the parameter for generating a random partition network, specifying the probability of presence of every possible contact between individuals from different partitions.

- `m`: [*OPTIONAL] (`int`)
  Required if `model` is specifed as `BA`. It is the parameter for generating a Barabási–Albert network (Number of contacts to attach from a new individual to existing individuals when creating this network).

- `random_seed`: [*OPTIONAL] (`int`)
  The value that help generate random numbers in the module. Using the same `random_seed` ensures identical output when the module is rerun with the same parameters.

In the upcoming sections, we introduce logistics and examples about the generation of various host contact networks.

2.1.1 User input

If the user opts to supply their own contact network, they need to make sure the contact network file is in an adjacency list format, which is a space-delimited file. Lines starting with `#` will be marked as comments. A typical command to run this mode is:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize ${HOST_SIZE} \
    -method user_input \
    -path_network ${NETWORK_PATH}
```

Listing 2.2: NetworkGenerator user_input mode

NetworkGenerator will read in the network from the file path user provided and check its format. If the file satisfies the format requirements, the network will be rewritten into a file named `contact_network.adjlist` in the working directory. Note that *OutbreakSimulator* cannot be run without a properly formatted and named `contact_network.adjlist` in the working directory. The step is recommended even if you're providing your own contact network. However, you can modify `contact_network.adjlist` by adding/deleting new contact edges manually as well, but we recommend rerunning *NetworkGenerator* by taking the absolute path of `${WKDIR}/contact_network.adjlist` as input for `-path_network` after your manual modification to make sure it is in a proper format except for non-canonical usage.

Non-canonical usage includes using weighted graph; by default the contact network is unweighted. Any connection between two hosts will only appear once in the file `contact_network.adjlist`. In the main module of e3SIM, each connection will only be evaluated once for transmission per tick. Users can add weight to an edge by duplicating the specific node ids in the relevant lines of `contact_network.adjlist`, so that the same contact will be evaluated multiple times per tick, imitating adding weight to an edge. Do not rerun the program or the weight of edges will not be retained, but take care to keep proper formatting when making this edit.

2.1.2 Well-mixed Host population

To generate a contact network for a specified host population size and a well-mixed host population, where every individual having the same number of expected contacts, use the Erdős–Rényi (ER) random graph model. The node degrees of the generated network will be normally distributed. The ER model is parameterized by p_{ER}, which represents the probability of any possible edge being present in the network. The expected number of contacts per individual can be calculated as $p_{ER} \times$ host size.

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize ${HOST_SIZE} \
    -method randomly_generate \
    -model ER \
    -p_ER ${PROB_ER}
```

Listing 2.3: NetworkGenerator ER network

NetworkGenerator will use the parameters provided to generate an Erdős–Rényi graph and write down the network into a file named `contact_network.adjlist` in the working directory. For examples to generate a uniform host population of $10,000$ hosts where each host has an 10 expected connections $(10 = 10000 * 0.001)$, we run:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize 10000 \
    -method randomly_generate \
    -model ER \
    -p_ER 0.001
```

Listing 2.4: NetworkGenerator ER network example

2.1.3 Superspreaders

Using this option will generate a host contact network using the Barabási–Albert (BA) random network model that features unbalanced connectivity, where a few potential "superspreaders" have high connectivity and the rest of the population have low connectivity, representative of a real-world social network for sexually transmitted disease, like HIV. The BA model has a single parameter, m.

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize ${HOST_SIZE} \
    -method randomly_generate \
    -model BA \
    -m ${m}
```

Listing 2.5: NetworkGenerator BA network

NetworkGenerator will use the parameters provided to generate an Barabási–Albert random network model and write down the network into a file named `contact_network.adjlist` in the working directory. For examples, to generate a uniform host population of $10,000$ hosts and has $m = 2$, we run:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize 10000 \
    -method randomly_generate \
    -model BA \
    -m 2
```

Listing 2.6: NetworkGenerator BA network example

2.1.4 Two Sub-populations

This option is designed for a host contact network with sub-population structures, where there are two partitions in the whole graph. Each partition has their own intra-partition contact probability and they share the same inter-partition contact probability. This sub-population configuration can represent separate rural and urban areas, two countries, an ecological barrier in natural world, and so on. A typical command for this option is:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize ${HOST_SIZE} \
    -method randomly_generate \
    -model RP \
    -rp_size ${POP1_SIZE} ${POP2_SIZE} \
    -p_within ${POP1_PROB} ${POP2_PROB} \
    -p_between ${PROB_BETWEEN_POP1_POP2}
```

Listing 2.7: NetworkGenerator RP network

NetworkGenerator will use the parameters provided to generate a random partition graph and write down the network into a file named `contact_network.adjlist` in the working directory. For example, the following command generates a host population of $10,000$ individuals, of which 6000 hosts come from an urban area where expected connectivity for each host is high as $12 (= 6000 * 0.002)$, and the rest 4000 come from a rural area where expected connectivity for each host is as low as $4 (= 4000 * 0.001)$. Meanwhile, there are a few connections between the two area so that hosts from one partition are expected to be connected with $3 (= 6000 * 0.0005)$ hosts from the other partition:

```
python network_generator.py \
    -wkdir ${WKDIR} \
    -popsize 10000 \
```

```
    -method randomly_generate \
    -rp_size 4000 6000 \
    -p_within 0.001 0.002 \
    -p_between 0.0005
```

Listing 2.8: NetworkGenerator RP network example

2.2 SeedGenerator

Initial sequences (a.k.a. seeds) are pathogen genomes that are planted in the host population at the start of the simulation of the main module. They can have distinct genomic sequences and different trait values such as transmissibility and drug-resistance. These trait values are directly affected by their genomic sequences (2.3). If the user wants simulations to seed the main module with the reference genome(s) instead of mutated genome(s) for each initial sequences compared to the reference genome, this module can be skipped. Note that using the reference genome as the initial sequence has to be specified when running *OutbreakSimulator* (Chapter 3). Otherwise, user can run *SeedGenerator* in random generation mode to generate mutated pathogen genomes by forward-in-time simulation, which serve as initial sequences in main simulator. If users choose to provide their own initial sequences, it is recommended to run *SeedGenerator* in user input mode to check the format of the user-provided genomic sequences (in `VCF` format).

SeedGenerator will create a new folder named `originalvcfs` under the provided working directory and store one VCF file for each initial sequence, named `seed.X.vcf` in the `originalvcfs` folder (`X` ranges from 0 to `${NUM_INIT_SEQ}` − 1). Please note that any existing `originalvcfs` folder in the working directory will be overwritten when running *SeedGenerator*.

SeedGenerator also stores the genealogy of the initial sequences in the working directory if applicable. If random generation mode is being used, it is not guaranteed that the generated seeds have a single common ancestor. If there is a single common ancestor, the genealogy will be stored in a `NWK` format file named `seeds.nwk` in the working directory; otherwise, *SeedGenerator* will create a folder named `seeds_phylogeny_uncoalesced` in the working directory and write a `NWK` file for each common ancestor for seeds. In the `NWK` file(s), tip labels of the tree are integer numbers from 0 to `${NUM_INIT_SEQ}` − 1, corresponding to the initial sequences' ID. The `seeds.nwk` file, if present in the working directory, will be used by the post-simulation modules to generate a complete genealogy of the sampled pathogens during the main simulation module. Please see more details in Chapter 3.

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq ${NUM_INIT_SEQ} \
    -method ${METHOD} \
    [Other params]
```

Listing 2.9: SeedGenerator Usage

We now introduce all the options:

- `wkdir`: [*REQUIRED] (`string`)
  Absolute path to the working directory, which should be consistent for each module in one workflow.
- `num_init_seq`: [*REQUIRED] (`integer`)
  Number of initial sequences for the main simulator, i.e. number of infected hosts at the start of the main simulation. Note that this quantity is for the main simulator, rather than the number of starting sequences for running the current module.
- `method`: [*REQUIRED] (`string`) `user_input` or `SLiM_burnin_WF` or `SLiM_burnin_epi`
  Method used for generating the initial sequences, which can only be one of the three choices: `user_input`

or SLiM_burnin_WF or SLiM_burnin_epi. By specifying user_input, users are required to provide their own VCF file for the initial sequences by -seed_vcf. By specifying one of the SLiM-burnin methods, you are required to provide a reference genome by -ref_path and other corresponding parameters for the method you choose.

- init_seq_vcf: [*OPTIONAL] (string)
  Required if method is specified as user_input. Absolute path to a VCF file containing the mutation profiles of the initial sequences.

- path_init_seq_phylogeny: [*OPTIONAL] (string)
  Optional if method is specified as user_input, not required otherwise. Absolute path to the initial sequences' phylogeny in NWK format.

- ref_path: [*OPTIONAL] (string)
  Required if method is specified as SLiM_burnin_WF or SLiM_burnin_epi. Absolute path to the reference genome of the pathogen you want to simulate with.

- use_subst_matrix: [*OPTIONAL] (bool)
  Optional if method is specified as SLiM_burnin_WF or SLiM_burnin_epi. Whether to use a substitution probability matrix to parametrize mutation probability, default is False. When False, the substitution model will be default to a Jukes–Cantor model.

- mu: [*OPTIONAL] (float)
  Required if method is specified as SLiM_burnin_WF or SLiM_burnin_epi, and that -use_subst_matrix is specified to be False. Mutation probability of the genomes during burn-in, which is the expected number of mutations per site for each genome in one tick, which is the parameter α for a Jukes–Cantor model (3.2.2). mu need not have the same value when running *OutbreakSimulator*, but note that this will cause the branch lengths of the generated phylogenies to be measured in different units, which may be misleading if directly compared. Manual re-scaling of the initial sequences' phylogeny might be of interest after running *SeedGenerator* in this case.

- mu_matrix: [*OPTIONAL] (str)
  ```
  '{"A": [0, p_ac, p_ag, p_at], "C": [p_ca, 0, p_cg, p_ct],
  "G": [p_ga, p_gc, 0, p_gt], "T": [p_ta, p_tc, p_tg, 0]}'
  ```
  Required if method is specified as SLiM_burnin_WF or SLiM_burnin_epi, and that -use_subst_matrix is specified to be True. A JSON string having the format shown above, where p_ij gives the probability of one site in allele i mutating to allele j in one tick. Following the tradition of SLiM, p_ii=0 for $i \in \{A,C,G,T\}$, but in practice the probability for a site in allele i staying in allele i during one tick will be calculated as $1 - \sum_{j \neq i}$ p_ij. For details of the evolution model, please refer to Section 3.2.2.

- n_gen: [*OPTIONAL] (int)
  Required if method is specified as SLiM_burnin_WF or SLiM_burnin_epi. Number of ticks/generations that will be run during the seed generation.

- Ne: [*OPTIONAL] (int)
  Required if method is specified as SLiM_burnin_WF. The effective population size of a Wright–Fisher model.

- host_size: [*OPTIONAL] (int)
  Required if method is specified as SLiM_burnin_epi. Host population size, which needs to be consistent with the size of the network in the contact_network.adjlist file in the working directory provided by -wkdir.

- `seeded_host_id`: [*OPTIONAL] (`list of ints`)
  Required if `method` is specified as `SLiM_burnin_epi`. ID(s) of the host(s) that will be infected with a pathogen whose genome is identical to the reference genome at first tick of the seed generation process. The host IDs have to be chosen from 0 to host_size - 1.

- `S_IE_prob`: [*OPTIONAL] (`float`)
  Required if `method` is specified as `SLiM_burnin_epi`. The probability of a successful transmission for one effective contact (an infected host and a susceptible host that are connected to each other) per tick. The default value is 0.

- `E_I_prob`: [*OPTIONAL] (`float`)
  Required if `method` is specified as `SLiM_burnin_epi` and `-latency_prob` is also greater than 0. The probability of an exposed host becoming infected per tick. The default value is 0.

- `E_R_prob`: [*OPTIONAL] (`float`)
  Could be supplied if `method` is specified as `SLiM_burnin_epi` and `-latency_prob` is greater than 0. The probability of an exposed host recovering per tick. The default value is 0.

- `latency_prob`: [*OPTIONAL] (`float`)
  Could be supplied if `method` is specified as `SLiM_burnin_epi`. The probability of a susceptible host becoming exposed upon transmission per tick. The default value is 0, meaning that all susceptible hosts become infected upon a relevant transmission.

- `I_R_prob`: [*OPTIONAL] (`float`)
  Required if `method` is specified as `SLiM_burnin_epi`. The probability of an infected host recovering per tick. The default value is 0.

- `I_E_prob`: [*OPTIONAL] (`float`)
  Required if `method` is specified as `SLiM_burnin_epi`. The probability of an infected host deactivating (transitioning to exposed state) per tick. The default value is 0.

- `R_S_prob`: [*OPTIONAL] (`float`)
  Required if `method` is specified as `SLiM_burnin_epi`. The probability of a recovered host losing immunity (going back to susceptible) per tick. The default value is 0, meaning that infected hosts will not go back to the susceptible state. Note that if this variable is zero, the outbreak will eventually diminish, which might cause insufficient number of seeding sequences to sample from.

- `random_seed`: [*OPTIONAL] (`int`)
  The value that help generate random numbers in the module. Using the same `random_seed` ensures identical output when the module is rerun with the same parameters.

2.2.1 User input

If the user wants to provide their own initial sequences' mutation information, they needs to make sure the provided file is in VCF format. The `.vcf` file should contain the exact number of samples as specified by `-num_seq_init`, and sample names do not matter, as initial sequences will be re-named based on the order in the provided `.vcf` file from 0 to `NUM_INIT_SEQ` − 1. If the provided `.vcf` file has more samples than what is specified by `-num_seq_init`, *SeedGenerator* will take the first `NUM_INIT_SEQ` columns for the initial sequences' genetic information. Please note that since e3SIM only supports haploid pathogens for now, the phasing won't be informative in the user-provided `.vcf` file. i.e. `0/1` and `1/0` or `1/1` in the sample columns will be all rewritten as `1` during validation. To run this mode:

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq ${NUM_INIT_SEQ} \
    -method user_input \
    -path_init_seq_phylogeny ${PATH_TO_INIT_SEQ_PHYLOGENY}
```

Listing 2.10: SeedGenerator Usage user_input

2.2.2 burn-in by Wright-Fisher model

The user can generate initial pathogen genomes with desired genetic diversity using the option to run a forward-in-time simulation using a Wright–Fisher model with SLiM. We run the WF model with `-Ne` constant population size for `-n_gen` generations with a Jukes–Cantor model for substitution model. Here is a template for how to initiate the seed generation process:

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq ${NUM_INIT_SEQ} \
    -method SLiM_burnin_WF \
    -ref_path ${REF_PATH} \
    -use_subst_matrix F \
    -mu ${MU} \
    -Ne ${Ne} \
    -n_gen ${NUM_GENS_BURNIN}
```

Listing 2.11: SeedGenerator Usage SLiM_burnin_WF

At the last generation/tick in *SeedGenerator*, initial sequences will be sampled from the current population. *SeedGenerator* will create a new folder named `originalvcfs` in the provided working directory and write one `.vcf` file for each initial sequence in the `$WKDIR/originalvcfs` directory named `seed.X.vcf` where $X \in \{0, \cdots, \$\texttt{NUM\_INIT\_SEQ} - 1\}$. *SeedGenerator* records the phylogeny of the sequences sampled, but sometimes not all sampled seeding sequences can be traced back to a single common ancestor. If one single common ancestor exists, the initial sequences' phylogeny will be written to the working directory as `seeds.nwk` in NWK format. If multiple ancestors exist, *SeedGenerator* will create a folder named `seeds_phylogeny_uncoalesced` in the working directory and write a `.nwk` file for each separate progeny.

For example, if we want to run a burn-in process for COVID-19 genomes using the WF model, we download COVID-19 genome from the NCBI website into the current directory, retrieving a file named `EPI_ISL_402124.fasta`. We specify the mutation rate as 1×10^{-6}, which corresponds to the mutation rate of COVID-19 per day in real-time scale. We can run the burn-in for 3650 ticks to simulate 10 years' burn-in period in real-time scale. Let us use an effective population size of 1000 and sample 5 initial sequences:

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq 5 \
    -method SLiM_burnin_WF \
    -ref_path EPI_ISL_402124.fasta \
    -mu 1e-6 \
    -Ne 1000 \
    -n_gen 3650
```

Listing 2.12: SeedGenerator Usage SLiM_burnin_WF example

2.2.3 burn-in by epidemiological model

Sometimes a burn-in process using the epidemiological model (which can be the same model and parameters as the real simulation) is desirable. in this case, the user must run *NetworkGenerator* program before they

proceed to this step. This mode can be viewed as a minimal version of the *OutbreakSimulator* with sampling only in the last generation/tick with no genetic effects being simulated. In this example, we use a customized substitution probability matrix for the evolution model by specifying `-use_subst_matrix T`.

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq ${NUM_INIT_SEQ} \
    -method SLiM_burnin_epi \
    -ref_path ${REF_PATH} \
    -use_subst_matrix T \
    -mu_matrix '{"A":[0,,,], "C":[,0,,], "G":[,,0,], "T":[,,,0] }' \
    -n_gen ${NUM_GENS_BURNIN} \
    -host_size ${HOST_SIZE}  \
    -seeded_host_id ${SEEDED_HOST_ID} \
    -S_IE_prob ${S_IE_prob} \
    -E_I_prob ${E_I_prob} \
    -E_R_prob ${E_R_prob} \
    -latency_prob ${latency_prob} \
    -I_R_prob ${I_R_prob} \
    -I_E_prob ${I_E_prob} \
    -R_S_prob ${R_S_prob}
```

Listing 2.13: SeedGenerator Usage SLiM_burnin_epi

Note that the transition probabilities all have a default value of 0 and are thus not strictly required to be otherwise specified. However, not providing non-zero probabilities for some transitions can lead to undesired effects. You often want to avoid making any state an absorbing dead end since that can lead to an epidemic dying out before any sequence get sampled. If the number of existing pathogens is less than the desired quantity of initial sequences, the program will instruct you to rerun the burn-in process or modify the parameters. The SEIR trajectory during the burn-in process is recorded and written as a compressed `.csv` file in the working directory provided by `-wkdir`, so users are welcome to inspect the file (named `burn_in_SEIR_trajectory.csv.gz`) to get an impression of the dynamics produced by your current rate parameter settings. If sufficient pathogens are present, *SeedGenerator* will create a new folder named `originalvcfs` under the provided working directory and write one `seed.X.vcf` file for each initial sequence.

As with the `SLiM_burnin_WF` option, *SeedGenerator* records the phylogeny of the sequences sampled, but sometimes not all sampled seeding sequences can be traced back to a single common ancestor. If one single common ancestor exists, the initial sequences' phylogeny will be written to the working directory as `seeds.nwk` in NWK format.

For example, if we want to run a burn-in process for COVID-19 genomes using an epidemiological model, we download COVID-19 genome from the NCBI website into the current directory, retrieving a file named `EPI_ISL_402124.fasta`. We want to specify a mutation probability matrix as such:

	A	C	G	T
A	–	7.16×10^{-8}	2.84×10^{-7}	5.45×10^{-8}
C	1.17×10^{-7}	–	3.51×10^{-8}	1.5×10^{-6}
G	4.32×10^{-7}	3.27×10^{-8}	–	2.32×10^{-7}
T	5.05×10^{-8}	8.54×10^{-7}	1.42×10^{-7}	–

We can run the burn-in for 3650 ticks to simulate 10 years' burn-in period in real-time scale. Say that we used *NetworkGenerator* to generate a contact network with 10000 hosts and decide that we want to start seed simulation at individual 256. We want to specify an epidemiological model where each infected host infects each of its connections by a probability of 0.03, and all infected hosts will go through a latent stage upon infection. Since the latency period for COVID-19 is usually 6 days after infection, we choose an activation probability $0.16(\approx \frac{1}{6})$. Since the recovery period for COVID-19 is usually 28 days after infection, we choose

a recovery probability of $0.035(\approx \frac{1}{28})$. Post-recovery immunity to COVID-19 lasts for up to 6 months but is highly protective only for the first 2 months, so we set 2 months as the expected time for losing immunity by choosing the probability of immunity loss to be $0.018(\approx \frac{1}{56})$. We do not want to allow infected hosts to go back to latency, or let latent hosts recover. Finally, we want to sample 5 initial sequences. To realize the specification, we run the following:

```
python seed_generator.py \
    -wkdir ${WKDIR} \
    -num_init_seq 5 \
    -method SLiM_burnin_epi \
    -ref_path EPI_ISL_402124.fasta \
    -use_subst_matrix T \
    -mu_matrix '{"A": [0,7.16e-08,2.848e-07,5.45e-08], "C": [1.17e-07,0,3.51e-08,1.5e
    -06], "G": [4.32e-07,3.27e-08,0,2.32e-07], "T": [5.05e-08,8.54e-07,1.42e-07,0]}'
    \
    -n_gen 3650 \
    -host_size 10000 \
    -seeded_host_id 256 \
    -S_IE_prob 0.003 \
    -E_I_prob 0.16 \
    -E_R_prob 0  \
    -latency_prob 1 \
    -I_R_prob 0.035 \
    -I_E_prob 0  \
    -R_S_prob 0.018
```

Listing 2.14: SeedGenerator Usage SLiM_burnin_epi example

For `-seeded_host_id`, you can manually check your contact network file to find a host that you think is good, or you can utilize *HostSeedMatch* program (Section 2.4) to find a specific host that satisfy your need.

2.3 GeneticEffectGenerator

GeneticEffectGenerator module in e3SIM is designed to configure the causal genomic elements for the main simulator module. *GeneticEffectGenerator* will create `causal_gene_info.csv` in the working directory, where each row represents one causal genomic element. The first three columns give the name, starting position, and ending position of each region. The other columns represent the effect sizes for mutations located in each causal genomic element for each trait. Effect size columns for transmissibility and drug resistance are named as `eff_size_transmissibilityX` or `eff_size_drX`, where `X` is the ID of the trait in each trait type. For example, this is a `causal_gene_info.csv` generated by *GeneticEffectGenerator*, where three causal genomic elements where mutations confer positive effect sizes to three different traits respectively are specified.

```
gene_name,start,end,eff_size_transmissibility1,eff_size_dr1,eff_size_dr2
segment1,450,550,0.0,0.3,0.0
segment2,650,1000,0.2,0.0,0.0
segment3,4300,4600,0.0,0.0,0.3
```

Listing 2.15: Example of an output file of GeneticEffectGenerator

As shown in the example above, *GeneticEffectGenerator* supports more than one trait for transmissibility and/or drug resistance, like `eff_size_dr1` and `eff_size_dr2`. e3SIM operates in an epoch manner (Section 4.1), so that within one epoch, only one of the traits for the same trait type could be utilized. In this example, during one epoch, users could activate drug resistance trait by using either `eff_size_dr1` or `eff_size_dr2`, which could differ among epochs. For details about how to specify this in the main module, please refer to Section 4.1.

The causal genomic element configuration file generated by *GeneticEffectGenerator* could come from two modes, user-defined or randomly generated. For a user-defined architecture, users are recommended to run

the user-defined mode to check the format of the file they provided. To randomly generate a genetic architecture, a GFF (general feature format) file where each item does not overlap with each other needs to be provided. By running *GeneticEffectGenerator*, the trait values for all seeding sequences, based on their mutation profiles and genetic architecture will be calculated and stored as a file named `seeds_trait_values.csv` in the working directory, which could be inspected by the user manually before running the next module.

During the execution of the main simulator module, if the causal genomic elements has net positive effect sizes, the trait value is expected to increase as new mutations accumulate at a specified mutation rate. To help users choose a suitable effect size that does not inflate the trait values to an unexpected degree, e3SIM includes an optional heuristic approach for normalizing the expected average trait value to a user-defined value by the end of the simulation by specifying `-normalize T`. For details of how to calculate the trait values and the normalization scheme, please refer to Section 4.3.

A typical command to run *GeneticEffectGenerator* is:

```
python genetic_effect_generator.py \
    -wkdir ${WKDIR} \
    -method ${METHOD} \
    [Other params]
```

Listing 2.16: GeneticEffectGenerator Usage

We now introduce all the options:

- `wkdir`: [*REQUIRED] (`string`)
  Absolute path to the working directory, which should be the same for all modules.
- `method`: [*REQUIRED] (`string`) `user_input` or `randomly_generate`
  The method of generating effect sizes, which can only be `user_input` or `randomly_generate`. By specifying `user_input`, users are required to provide their own effect size file via `-effsize_path`, that has the format shown in code block 2.15. By specifying `randomly_generate`, you are required to provide the number of traits you want to generate (`-trait_n`) and other relevant parameters.
- `effsize_path`: [*OPTIONAL] (`string`)
  Required if `method` is specified as `user_input`. Absolute path to the user-provided effect size file.
- `gff`: [*OPTIONAL] (`string`)
  Required if `method` is specified as `randomly_generate`. Absolute path to the GFF file annotates the genome.
- `trait_n`: [*REQUIRED] (`JSON string`) `'{"transmissibility": x, "drug-resistance": y}'`
  A `JSON` string specifying the number of transmissibility and drug resistance traits. `x` and `y` are integer numbers specifying the number of traits for each trait type.
- `causal_size_each`: [*OPTIONAL] (`list of ints`)
  Required if `method` is specified as `randomly_generate`. It represents the number of causal genomic elements that should be chosen from the `.gff` file for each trait. It is a list of integers that has the length of the sum of `-trait_n`. For example, if 1 transmissibility trait and 2 drug resistance traits are desired, an integer list of length 3 should be provided in this option, with the first integer representing the number of causal genomic elements for the transmissibility trait, the second and third integers representing the number of causal genomic elements for the two drug resistance traits.
- `es_low`: [*OPTIONAL] (`list of floats`)
  Required if `method` is specified as `randomly_generate`. Lower bound on the effect size of each causal genomic element for each trait. It has to be a numeric list that has the same length of the sum of `-trait_n`.

- `es_high`: [*OPTIONAL] (`list of floats`)
  Required if `method` is specified as `randomly_generate`. Upper bound on the effect size of each genetic region for each trait (transmissibility and drug resistance) and thus has to be a numeric list that has the same length of the sum of `-trait_n`.

- `normalize`: [*OPTIONAL] (`bool`)
  Whether to adjust the original effect sizes in order to normalize the expected average trait value to a user-defined value by the end of the simulation. Default is `False`.

- `use_subst_matrix`: [*OPTIONAL] (`bool`)
  Optional if `normalize` is specified as `True`. Whether to use a substitution probability matrix to parametrize mutation probability in the main simulation module, default is False.

- `sim_generation`: [*OPTIONAL] (`list of ints`)
  Required if `method` is specified as `randomly_generate` and `normalize` is specified as `True`. Number of ticks that is expected to be run in *OutbreakSimulator*.

- `mut_rate`: [*OPTIONAL] (`list of floats`)
  Required if `normalize` is specified as `True` and `-use_subst_matrix` is specified to be False. Mutation probability of the genomes during burn-in, which is the expected number of mutations per site for each genome in one tick, which is the parameter α for a Jukes–Cantor model (3.2.2).

- `mu_matrix`: [*OPTIONAL] (`str`)
  `'{"A": [0, p_ac, p_ag, p_at], "C": [p_ca, 0, p_cg, p_ct], "G": [p_ga, p_gc, 0, p_gt], "T": [p_ta, p_tc, p_tg, 0]}'`
  Required if `normalize` is specified as `True` and `-use_subst_matrix` is specified to be True. A `JSON` string having the format shown above, where `p_ij` gives the probability of one site in allele `i` mutating to allele `j` in one tick. Following the tradition of SLiM, `p_ii=0` for $i \in \{\text{A,C,G,T}\}$, but in practice the probability for a site in allele `i` staying in allele `i` during one tick will be calculated as $1 - \sum_{j \neq i} \texttt{p\_ij}$. For details of the evolution model, please refer to Section 3.2.2.

- `final_T`: [*OPTIONAL] (`numerical`)
  Optional if `normalize` is specified as `True`. The expected average trait value for all seeding sequences by the end of the main simulator module. Default is 1.

- `ref`: [*OPTIONAL] (`string`)
  Required if `normalize` is specified as `True` and `-use_subst_matrix` is specified to be True. Absolute path to the reference genome of the pathogen you want to simulate with.

- `random_seed`: [*OPTIONAL] (`int`)
  The value that help generate random numbers in the module. Using the same `random_seed` ensures identical output when the module is rerun with the same parameters.

2.3.1 User input

By specifying `-method user_input`, user could provide a customized genetic architecture file. *GeneticEffectGenerator* will read the user-provided effect size file, check the format and entry of the file, and then rewrite it to `$WKDIR/causal_gene_info.csv`. *GeneticEffectGenerator* also calculates the trait values for each seed based on the genetic architecture file provided, and stores the trait values in `$WKDIR/seeds_trait_values.csv`. The provided genetic architecture file has to be in the format shown in Code block 2.15 with identifiable column names.

```
python genetic_effect_generator.py \
    -wkdir ${WKDIR} \
    -method user_input \
    -effsize_path ${EFF_SIZE_PATH}
```

Listing 2.17: GeneticEffectGenerator Usage user_input

2.3.2 Generate from GFF file

The user can also randomly generate an effect size file using a genome annotation file as input. A genome annotation file in GFF format for the reference genome can be downloaded from NCBI. Since e3SIM does not permit overlapping causal genomic elements (For example, having one genomic element ranging from position 100 to 200 and another genomic element ranging from position 250 to 250 would cause error), please make sure that all entries in the GFF file you provided to *GeneticEffectGenerator* do not overlap with each other in terms of positions. The GFF doesn't have to be the exact one downloaded from NCBI, users can also divide the genome into segments arbitrarily and provide a GFF format of the customzied genome annotation file.

The genetic architecture for one trait is a set of causal genomic regions along with their effect sizes for the selected for the trait. Several traits for one trait type could be specified. When executing *GeneticEffectGenerator* in the random generation mode, users are required to specify numbers of traits for the two different trait types: transmissibility and drug resistance according to the needs in the main simulator. For example, two traits for drug resistance with different causal genomic regions and effect sizes could be generated, representing resistance for two different types of treatments. In *OutbreakSimulator*, the user could specify two epochs with those two different treatments by activating different drug resistance traits for different epochs.

For each trait, the number of causal genomic elements, the lower bound, and the upper bound need to be specified by `-causal_size_each`, `-es_low` and `-es_high`. Firstly, specific number of causal genomic elements are uniformly drawn from the entries of the GFF file. Secondly, each effect size is uniformly drawn between the two bounds for each causal genomic element of the trait. Thirdly, if `-normalize T` is specified, the effect sizes will be re-scaled based on the mutation rate and number of ticks expected to run in the main simulator. As a result, the expected average trait values for the pathogen genomes at the end of the simulation will be a user specified value (`-final_T`). Lastly, if *SeedGenerator* has been run before (The `originalvcfs` folder was detected in the working directory), *GeneticEffectGenerator* calculates the current trait values for all seeds based on the genetic architecture specified and stores it in the working directory. To run genetic effect generators without normalization:

```
python genetic_effect_generator.py \
    -wkdir ${WKDIR} \
    -method randomly_generate \
    -gff ${GFF_PATH} \
    -trait_n
    '{"transmissibility": ${NUM_SET_TRANSMISSIBILITY}, "drug_resistance": ${
    NUM_SET_DRUGRESIST}}' \
    -causal_size_each ${NUM_GENE_SET_1} ... ${NUM_GENE_SET_N} \
    # N = ${NUM_SET_TRANSMISSIBILITY}+${NUM_SET_DRUGRESIST}
    -es_low ${LOW_GENE_SET_1} ... ${LOW_GENE_SET_N} \
    -es_high ${HIGH_GENE_SET_1} ... ${HIGH_GENE_SET_N}
    [other params when normalize = T]
```

Listing 2.18: GeneticEffectGenerator Usage randomly generate

For example, now we want to generate the genetic architecture for simulating a *Mtb* outbreak. *Mtb* has around 4000 genes, among which we select 5 of them to be causative to higher transmissibility, and we choose 3 to be causative to higher drug-resistance. We want to have two sets of drug-resistance traits, so that we can simulate two different treatment stages later. We want to generate effect sizes between 0 and 5 for

transmissibility. For the first drug-resistance trait set, we want the range of effect sizes to be smaller, so we choose 1 and 3 as bounds. For the second drug-resistance trait set, we want the range to be bigger, so that we can get very different effect sizes for different genetic regions - we choose 0.1 and 7 as bounds. We want to normalize all the effect sizes, since we are running the simulation for 1000 generations/ticks with a mutation rate of 4.4×10^{-8}. This is because we do not want the transmissibility to inflate too fast. To run this, we download the `.gff` file from the NCBI website for the TB genome, and filtered to get a gene-only `GFF` file which does not contain overlapping entries: `GCF_000195955.2_ASM19595v2_genomic.overlap.gff`. We then execute:

```
python genetic_effect_generator.py \
    -wkdir ${WKDIR} \
    -method randomly_generate \
    -gff GCF_000195955.2_ASM19595v2_genomic.overlap.gff \
    -trait_n  '{"transmissibility": 1, "drug_resistance": 2}' \
    -causal_size_each 5 3 3 \
    -es_low 0 1 0.1 \
    -es_high 5 3 7 \
    -normalize true \
    -sim_generation 1000 \
    -mut_rate 4.4e-8
```

Listing 2.19: GeneticEffectGenerator Usage randomly generate example

The generated causal gene information file can be inspected and manually modified by users as well. It is recommended to rerun by `user_input` mode after modifications, so that trait values for each seed can be recalculated. Please note that it is possible that one causal genomic element has non-zero effect sizes for more than one trait, which provides a framework to simulate **pleiotropy**. This cannot be deliberately specified in random generation, but it is a potential outcome, and users can also manually change the effect file to reach this effect. The effect sizes can also be negative to simulate a negative effect on some traits. The user can exploit this possibility to define a gene in which the mutations induce lower transmissibility and higher drug-resistance at the same time. During the main simulator module, a maximum trait value could be set for each trait type (Section 3).

2.4 HostSeedMatcher

After configuring the genetic architectures of the pathogen genomes, the last step before running the main simulator module is to determine the start of the simulation, which is, determining which hosts are patient zero(s) of the simulated outbreak, and which seeding sequence each patient zero is infected by. *HostSeedMatcher* provides different methods to match the seeding sequences to hosts. A typical command to run *HostSeedMatcher* is:

```
python seed_host_match_func.py \
    -wkdir ${WKDIR} \
    -method ${METHOD} \
    -num_init_seq ${NUMBER_OF_INITIAL_SEQUENCES} \
    [Other params]
```

Listing 2.20: HostSeedMatch Usage

Upon successful execution of this module, one `csv` file named `seed_host_match.csv` is produced in the working directory, which has the format of: 1) two columns named `host_id` and `seed`; 2) each row representing one pair of seed-host match; 3) all seeds must be matched to one host; 4) The rows are ordered in the order of the ID of hosts. One example of the output file is:

```
seed,host_id
4,1188
```

```
3,1290
1,4059
0,4446
2,7100
```

Listing 2.21: Example of an output file of HostSeedMatcher

Five seeding sequences are provided in this file, each is matched to one of the hosts. The seeds are ordered as $4, 3, 1, 0, 2$ due to the ID of the hosts they are matched to. The host IDs that are smaller appear earlier, which is required for the correct execution of the main simulator module.

The available options for running *HostSeedMatcher* are as follows:

- `wkdir`: [*REQUIRED] (`string`)
  Absolute path to the working directory, which should consistent with the directory used for running previous modules. It is assumed that *NetworkGenerator* has been run, so that `contact_network.adjlist` file is in this directory.

- `method`: [*REQUIRED] (`string`) `user_input` or `randomly_generate`
  The method used matching, which can only be `user_input` or `randomly_generate`. When specifying `user_input`, users are required to provide their own matching file using the `-path_matching` option. On the other hand, specifying `randomly_generate` necessitates providing a match scheme and matching parameters. (`-match_scheme` and `-match_scheme_param`).

- `num_seq_init`: [*REQUIRED] (`int`)
  Total number of seeding sequences to be matched to the hosts.

- `path_matching`: [*OPTIONAL] (`string`)
  Required when `method` is set to `user_input`. Absolute path to the user-provided matching file, which should have the format shown in Code block 2.21.

- `match_scheme`: [*OPTIONAL] (`JSON string`) `'{"0": M0, "1": M1 ...}'`
  Required when `method` is set to `randomly_generate`. A dictionary specifying the matching scheme for each initial sequence, having the format shown above. The keys are the IDs of the seeds, and the values (M0, M1...) are the matching scheme for each seeding sequence, respectively. The values (M0, M1...) should be one of `["ranking", "percentile", "random"]`. If all seeding sequences are to be matched randomly, the dictionary can be replaced by just a single "`random`". For seeding sequences that were not specified with a matching scheme, "`random`" will be the default.

- `match_scheme_param`: [*OPTIONAL] (`JSON string`) `'{"0": P0, "1": P1 ...}'`
  Required when `method` is set to `randomly_generate`. A dictionary containing parameters for the matching scheme of each seeding sequences. For example, if for seeding sequence 0, "`ranking`" is specified in `-match_scheme`, then `P0` here should be an integer number specifying the rank of the host. For detailed introduction, please refer to the sections below for the format of parameters corresponding to each matching method.

- `random_seed`: [*OPTIONAL] (`int`)
  The value that help generate random numbers in the module. Using the same `random_seed` ensures identical output when the module is rerun with the same parameters.

2.4.1 User-provided matching file input

If users would like to match host(s) with exact ID(s) to the seeding sequences, one customized seed-host matching file could be provided and checked by *HostSeedMatch* module in the `user_input` mode. The pro-

vided matching file has to be in specific format as described by Code block 2.21. An example for running *HostSeedMatch* in this mode is:

```
python seed_host_matcher.py \
    -wkdir ${WKDIR} \
    -method user_input \
    -num_init_seq 10 \
    -path_matching ${PATH_TO_MATCHING_CSV_FILE}
```

Listing 2.22: HostSeedMatch user input example

2.4.2 Generate matching based on user-specified methods & parameters

e3SIM provides three different matching schemes: `ranking`, `percentile` and `random`. Different matching scheme could be chosen for each seeding sequences, specified by the `-match_scheme` option. Separate matching scheme parameters have to be provided for each seeding sequences. All three schemes are explained below with an example:

Ranking Method (`'ranking'`)

- This scheme assigns seeds to specific hosts based on the host's contact degree ranking. The required parameter is one integer number denoting the host ranking. For example, the host with the highest number of contacts receives the first seed, the second-highest receives the second seed, and so on. Each seed can be assigned to a host with any rank.
- Lower ranks (smaller numbers) correspond to hosts with higher connectivity. For example, rank 1 corresponds to the host with the highest connectivity in the contact network.
- EXAMPLE: To assign seeding sequnece 1 to a host that ranks 100 in connectivity: `-match_scheme '{..., "1": "ranking", ...}', -match_scheme_param '{..., "1": 100, ...}'`

Percentile Method (`'percentile'`)

- This scheme assigns seeds to hosts within a connectivity range, specified by percentiles of the host contact degree distribution. The required parameter is an integer list of length two, specifying one percentile range to choose from. Thus the integer numbers in the list has to be within the closed interval [0, 100], with the first number smaller than the second number.
- All hosts are assigned to different percentiles based on the contact degree distribution. By specifying a percentile range, one host within that range is uniformly drawn to be the matched host.
- EXAMPLE: To assign seeding sequence 1 to a host whose connectivity is in the top 25%: `-match_scheme '{..., "1": "percentile", ...}', -match_scheme_param '{..., "1": [0, 25], ...}'`

Random Method (`'random'`)

- This scheme randomly select one host from the available host lists. No parameter is required for this scheme. It is also the default method for all seeding sequences. For each seeding sequence, if no specific matching scheme is provided, it is matched by this scheme.
- EXAMPLE: If users want to match all seeding sequences randomly, they don't need to provide `-match_scheme` and `-match_scheme_param` option.

We hereby show a practical example for executing `HostSeedMatcher`: Consider the following scenario that matches 3 seeding sequences by three different schemes. We want to match the first sequence to a host completely at random. We want to match the second sequence to the host possessing the highest number of contacts within the network. Finally, we want to match the third sequence with a host from that is less connected than the medium value. To achieve this customized instance of matching, the following command would be executed:

```
python seed_host_matcher.py \
    -wkdir ${WKDIR} \
    -num_init_seq 3 \
    -method randomly_generate \
    -match_scheme '{"0": "random", "1": "ranking", "2": "percentile"}'
    -match_scheme_param '{"1": 1, "2": [50, 100]}'
```

Listing 2.23: HostSeedMatch user specified matching method(s) example

Part II

Simulator

Chapter 3

Overview of the usage

Contents

3.1 Run OutbreakSimulator

In this chapter, we will introduce the main simulator module of e3SIM: *OutbreakSimulator*. To run this module, users should make sure that they have already run necessary pre-simulation modules. *OutbreakSimulator* requires the existence of several files with specific format **in the specified working directory** (e.g., `${WKDIR}`), which are generated by the pre-simulation modules. These files are listed below:

- `contact_network.adjlist`: Required. Generated by *NetworkGeneratror*, an adjacency list specifying the contact network of the host population (Section 2.1).
- `originalvcfs`: Required if the user would like to use seeding sequences different from the reference genome. Generated by *SeedGenerator*, a folder where the `VCF` files for all seeding sequences locates (The `VCF` files are named as `seed.X.vcf` where `X` is the ID of the seeding sequences starting from 0) (Section 2.2)
- `seed_host_match.csv`: Required. Generated by *HostSeedMatch*, a `CSV` file specifying the matching of seeding sequences to the hosts (Section 2.4).
- `seeds.nwk`: Optional. Generated by *SeedGenerator* (Section 2.2), a `NWK` file specifying the genealogy of the seeding sequences. If the file exists, in the post-simulation module, *OutbreakSimulator* could generate the full pathogen genealogy by binding the genealogy of the progeny of each seeding sequence to the the genealogy of the seeding sequences.
- `causal_gene_info.csv`: Required if the user would like to incorporate the coupling of evolutionary process and epidemiological processes in the simulation. Generated by *GeneticEffectGenerator*, a `CSV` file specifying the positions of causal genomic elements and their effect sizes for crucial traits (Section 2.3).
- A configuration file in `JSON` format: Required. If the user used the GUI to perform the pre-simulation modules, the configuration file should be present in the working directory. Otherwise, users are required to provide a customized configuration file to *OutbreakSimulator*, and the template to the configuration file is in our github repository.

To execute `OutbreakSimulator`, only one option is required, which is the absolute path to the configuration file that specifies all the parameters of the simulation. An example to run `OutbreakSimulator` is:

```
python outbreak_simulator.py \
    -config ${YOUR_SIM_CONFIG_PATH}
```

Listing 3.1: OutbreakSimulator usage

- `config` [*REQUIRED] (`string`)
  Absolute path to the configuration file of running *OutbreakSimulator* in a specific format.

We will explain the structure and contents of the configuration file in this chapter. However, it is recommended to go through Chapter 4, that breaks down the components of `OutbreakSimulator` and explains how the simulation works in detail in order to fully understand the meaning of the parameters. In Chapter 5, we will walk through some realistic examples on how to specify and simulate your own model.

3.2 Configuration file

There are several sections in the configuration file: `"BasicRunConfiguration"`, `"EvolutionModel"`, `"SeedsConfiguration"`, `"GenomeElement"`, `"NetworkModelParameters"`, `"EpidemiologyModel"` and `"Postprocessing_options"`. In the following paragraphs, we introduce the format and meaning for all options within the configuration file in detail.

3.2.1 BasicRunConfiguration

This section specifies basic configurations to execute the simulator.

- `"cwdir"`: (`string`)
  Absolute path to the working directory, which should be the same path provided to run all the pre-simulation programs, so it is expected that necessary files are present in this directory.
- `"n_replicates"`: (`int`)
  The number of replicates to run with the configuration specified in this file.
  (`OutbreakSimulator` will create sub-directories for the output of each replication in the working directory specified by `"cwdir"`).

3.2.2 EvolutionModel

This section specifies the evolution model used in the simulation.

- `"n_generation"`: (`int`)
  The number of ticks the user wants the simulation to run.
- `"subst_model_parameterization"`: (`string`) `mut_rate_matrix` or `mut_rate`
  How to parameterize the molecular evolution model. If `mut_rate` is specified, `"mut_rate"` needs to be specified in the configuration file. If `mut_rate_matrix` is specified, `"mut_rate_matrix"` needs to be specified in the configuration file.
- `"mut_rate"`: (`float`)
  Needed to be specified when `"subst_model_parameterization"`is specified to be `mut_rate`. A single mutation probability each tick for any site in the pathogen genome. By using the `"mut_rate"` mode

for evolution model, `OutbreakSimulator` uses the Jukes–Cantor model in SLiM, thus, the probability of one site mutating into each of the alternative allele (for example, A to C) per tick will be `mut_rate`/3, and the overall mutation probability will be `mut_rate`.

- `"mut_rate_matrix"`: (`list`)
  `[[0,p_AC,p_AG,p_AT],[p_CA,0,p_CG,p_CT],[p_GA,p_GC,0,p_GT],[p_TA,p_TC,p_TG,0]]`
  Needed to be specified when `"subst_model_parameterization"`is specified to be `mut_rate_matrix`. A numerical list of length 4 showing the mutation probability matrix to be used in the simulation. The 4 sub-lists in the list represents the probability of mutation per tick for a site that has allele A,C,G,T. `p_uv` represents the probability of allele u mutating into allele v per tick. All the probabilities should be a numerical value within the closed interval [0,1], and the sum of the probabilities for each sub-list should not exceed 1.

- `"within_host_reproduction"`: (`bool`) `true` or `false`
  Whether to activate within-host reproduction. If activated, pathogens are permitted to reproduce within the same host in a probability specified by `"within_host_reproduction_rate"`.

- `"within_host_reproduction_rate"`: (`float`)
  Probability of reproducing within-host for each existing pathogen genome during each tick. This parameter will not be applied unless `"within_host_reproduction"` is set to `true`.

- `"cap_withinhost"`: (`int`)
  The maximum number of pathogens that is allowed to exist in one host. i.e., if the cap is reached for one host, then no within-host reproduction will happen for the host, and the host cannot be an infectee in a transmission event. The parameter is default to 1. However, if within-host reproduction and/or superinfection is specified to be true, it is recommended to set a different value for `"cap_withinhost"`.

3.2.3 SeedsConfiguration

This section specifies the initial sequences.

- `"seed_size"`: (`int`)
  Number of seeding pathogen sequences. Should be the same as what is used for *SeedGenerator* and *HostSeedMatcher*. For example, it is expected that the `originalvcfs` folder in the working directory contains the same number of `VCF` files if applicable.

- `"use_reference"`: (`bool`) `true` or `false`
  Whether to use the reference genome as the seeding sequence(s). If `true`, then it is not required to run *SeedGenerator* in the pre-simulation modules.

3.2.4 GenomeElement

This section specifies the causal genomic elements settings that is going to be used in the simulation.

- `"use_genetic_model"`: (`bool`) `true` or `false`
  Whether to let mutation profiles to affect trait values such as transmissibility and drug-resistance. If `true`, then it is required to run *GeneticEffectGenerator* in the pre-simulation modules.

- `"ref_path"`: (`string`)
  Absolute path to the reference genome in `FASTA` format. It should be the same as the one used for *SeedGenerator* if applicable. The `FASTA` file should contain only one sequence, and only composed by the four alleles A,C,G and T.

- "traits_num": (JSON string) '{"transmissibility": x, "drug-resistance": y}'
  A JSON string specifying the number of traits (the genetic architectures) for to 'transmissibility' and 'drug_resistance', which should be the same as what is used in *GeneticEffectGenerator*'s -trait_n option. e.g., "traits_num": '{"transmissibility": 1, "drug_resistance": 2}' represents 1 set for transmissibility and 2 sets for drug-resistance.

3.2.5 NetworkModelParameters

This section specifies information about the host population.

- "use_network_model": (bool) true
  Whether to specify a host contact network, should always be true in the current version.
- "host_size": (int)
  host population size, which should be the same as what was specified in *NetworkGenerator*.

3.2.6 EpidemiologyModel

This section specifies the epidemiological model, which includes the compartmental model and the transition probabilities between compartments.

- "slim_replicate_seed_file_path": (string)
  Absolute path to a .csv file with a single column named random_number_seed that stores the random number generator ("seeds" in computational science generally, different from the seeding sequences in e3SIM) for each replicate. If not specified, the random number generator will be chosen by SLiM, and could be checked afterwards in the logging file of SLiM.
- "model": (string) SIR or SEIR
  The compartmental model that will be applied. The user can choose from "SIR" or "SEIR". See Section 4.4 for the permitted compartments and transitions. The transitions involving the exposed state will be disabled if "SIR" is chosen.
- "epoch_changing":
  This subsection specifies the epoch setting of the simulation. Ticks, the time units in e3SIM, are organized into *epochs*. Each epoch comprises a sequence of consecutive ticks. Within each epoch, one set of epidemiological parameters and genetic architecture is used. Please see more details about epochs in Section 4.1.
  - "n_epoch": (int)
    Number of epochs used in the simulation. By specifying this parameter, all configurations in "genetic_architecture" and "transition_prob" section will be lists with the length of n_epoch, as one set of parameters needs to be specified for each epoch.
  - "epoch_changing_generation": (list of ints)
    The tick(s) at which the simulation switches to the next epoch and its corresponding set of parameters. It should be an integer list with the length of n_epoch - 1. For example, if "n_epoch": 3 is specified, then you should specify "epoch_changing_generation":[t1, t2], where $t1, t2 \in \{1, \cdots, \text{n\_generation}\}$ ("n_generation" was specified in Section 3.2.2).
- "genetic_architecture":
  This subsection specifies whether to invoke genetic architecture, and which trait to use in each epoch. Each parameter listed in this subsection should take the format of a numerical list whose length is the same to n_epoch.

  - `"transmissibility"`: (`list of ints`)
    An integer list that specifies which transmissibility trait should be used for each epoch. Should be a list of length `n_epoch`, and each element is an integer representing the trait ID. For example, if `"traits_num"` is specified to be `'{"transmissibility": 2, "drug_resistance": 3}'`, meaning that there are 2 traits for transmissibility, then here each element in `"transmissibility"` should be one of $\{0, 1, 2\}$, where 0 represents not using genetic effect on transmissibility in this epoch, and 1 or 2 denotes that the first or second transmissibility genetic architecture will be used in an epoch.
  - `"cap_transmissibility"`: (`list of floats`)
    The cap of transmissibility values in this epoch. Due to the nature of our simulation, transmissibility can inflate if all effect sizes are positive. You can set a cap for each epoch so that if transmissibility is calculated to be higher than the cap, then the cap value is being used. Should be a list of length `n_epoch`.
  - `"drug_resistance"`: (`list of int`)
    An integer list that specifies which drug resistance trait should be used for each epoch. Should be a list of length `n_epoch`, where each element is an integer representing the trait id. For example, if `"traits_num"` is specified to be `'"transmissibility": 2, "drug_resistance": 3'`, meaning that you have 3 effect size sets for drug-resistance, then here each element in `"drug_resistance"` should be one of $\{0, 1, 2, 3\}$, where 0 represents not using genetic effect on drug-resistance in an epoch, and 1 or 2 or 3 denotes that the first, second or third drug resistance genetic architecture will be used in an epoch.
  - `"cap_drugresist"`: (`list of floats`)
    The cap of drug resistance values in this epoch. Due to the nature of our simulation, drug-resistance can inflate if all effect sizes are positive. You can set a cap for each epoch so that if drug-resistance is calculated to be higher than the cap, then the cap value is being used. Should be a list of length `n_epoch`.

- `"transition_prob"`:
  This subsection specifies the (base) transition probability between compartments in each epoch. Each parameter listed in this subsection should take the format of a numerical list whose length is the same to `n_epoch`.

  - `"S_IE_prob"`: (`list of floats`)
    The base probability of a successful transmission for an effective contact in one tick. The actual probability of transmission for one contact could be modified by transmissibility trait of the pathogen from the infector if `"transmissibility"` isn't 0 in this epoch.
  - `"I_R_prob"`: (`list of float`)
    The base probability of an infected host recovering for each epoch. In epochs where `"drug_resistance"` isn't 0, it represents the probability of an infected host being evaluated for recovery per tick, and the actual recovery probability is modified by drug resistance trait of all the pathogens in the host.
  - `"R_S_prob"`: (`list of float`)
    The probability of a recovered host losing immunity per tick (going back to susceptible state) during each epoch.
  - `"latency_prob"`: (`list of float`)
    The probability of a susceptible host becoming exposed upon undergoing a successfull transmission for each epoch.
  - `"E_I_prob"`: (`list of float`)
    The probability of an exposed host activating (going to the infected state) per tick for each epoch.

  - "I_E_prob": (list of float)
    The probability of an infected host deactivating (going to the exposed state) for each epoch.
  - "E_R_prob": (list of float)
    The probability of an exposed host recovering per tick for each epoch.
  - "sample_prob": (list of float)
    The sequential sampling probability of an infected host per tick for each epoch.
  - "recovery_prob_after_sampling": (list of float)
    The probability of a host recovering upon being sequentially sampled for each epoch.

- "massive_sampling": A subsection that specifies concerted sampling events (if any). Please refer to ?? for how these events work.

  - "event_num": (int)
    Number of concerted sampling events to happen.
  - "generation": (list of int)
    Ticks when each concerted sampling event happens. Should be a list of length event_num.
  - "sampling_prob": (list of float)
    Probability of infected hosts being sampled in each concerted sampling event. Should be a list of length event_num.
  - "recovery_prob_after_sampling": (list of float)
    Probability of infected hosts recovering upon being sampled in each concerted sampling event. Should be a list of length event_num.

- "super_infection": (bool) true or false
  Whether to permit super-infections. If this option is set to true, the host can be an infectee regardless of existing infection, as long as the number of pathogens within the host does not exceed the specified maximum capacity ("cap_withinhost"), and all successful transmissions to this host within a single tick are retained.

3.2.7 Postprocessing_options

This subsection specifies whether to do post-simulation data processing and how to visualize the simulated data. Please refer to Part III for all post-processing details.

- "do_postprocess": (bool) true or false
  Whether to do post-simulation processing. If true, then some plots and metadata files will be generated. If false, then for each replicate, only logging files for epidemiological events, treesequence files of the sampled pathogen genomes will be generated.

- "tree_plotting":
  This sub-subsection specifies how to produce the plotted genealogy of sampled pathogen genomes.

  - "branch_color_trait": (int)
    How to color the branches in the generated tree plot, given that "do_postprocess": true. If specified as 0, branches will be colored by seed ID. If specified as IDs of other traits (both transmissibility and drug resistance traits are ordered together), branches will be colored by relative trait value (lowest being blue, highest being red). For example, if "traits_num" is specified to be '{"transmissibility": 2, "drug_resistance": 3}', then specifying "branch_color_trait": 2 will generate a tree plot with branches colored by the second transmissibility trait, while specifying "branch_color_trait": 4 will generate a tree plot with branches colored by the second drug resistance trait.

- `"heatmap"`: (`string`) `none` or `transmissibility` or `drug_resistance`
  Whether and which trait to plot as a heatmap for each sample in the transmission tree plot, given that `"do_postprocess": true` is specified.

Chapter 4

Modules of the simulator

Contents

4.1 Epochs and ticks

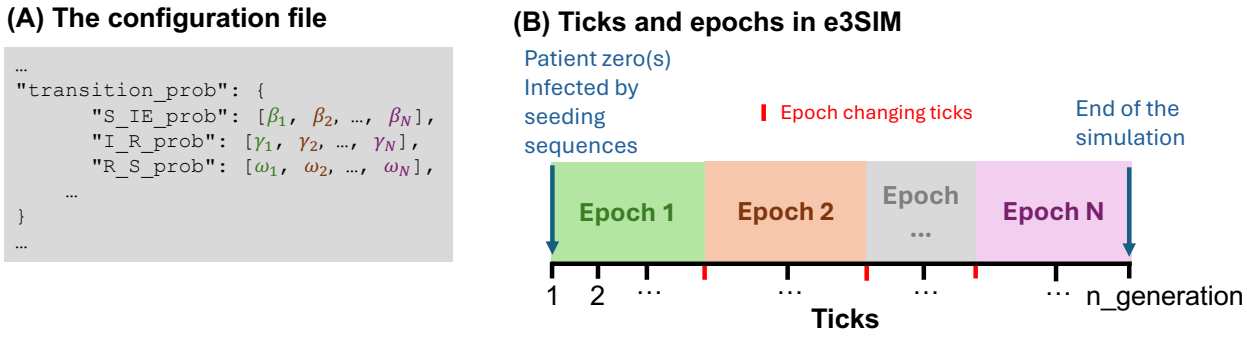


Figure 4.1: **Ticks and epochs in e3SIM.** (A) The configuration file for this simulation. (B) All the ticks are assigned into N epochs, and each epoch uses one set of epidemiological parameters, as colored in (A).

`OutbreakSimulator` operates in discrete time, where each step, or tick, represents the unit of time during which actions occur for each pathogen and host. The simulation concludes when a predefined number of ticks is reached. In every tick, decisions are made for all possible reproduction and compartmental transitions. Ticks are organized into epochs, each comprising a sequence of consecutive ticks. Within each epoch, the base rates for compartmental transitions and the genetic architecture for each trait remain constant. These parameters can change to new values when the simulation transitions to a new epoch. The epoch structure allows the simulation to incorporate time-dependent environmental changes, such as implementing new intervention strategies at specific time points after the outbreak starts. For example, in the following configuration file (only part of it is shown), two epochs are specified in the configuration file:

```
...
"epoch_changing": {
```

```
  "n_epoch": 2,
  "epoch_changing_generation": [100]
},
"genetic_architecture": {
  "transmissibility": [0, 1],
  "drug_resistance": [1, 2],
  ...
},
"transition_prob": {
  "S_IE_prob": [0.1, 0.2],
  "I_R_prob": [0.3, 0.3],
  ...
}
...
```

Listing 4.1: Config file for multiple epochs

In the first epoch (tick 1 through tick 99), no genetic architecture is specified for transmissibility trait, and the first drug resistance trait is used. The base transmission probability and base recovery probability are 0.1 and 0.3, respectively. In the second epoch (tick 100 through the end of the simulation), the first transmissibility trait is used, and the second drug resistance trait is used. The base transmission probability and base recovery probability are 0.2 and 0.3, respectively.

4.2 Evolutionary model

In e3SIM, mutations accumulate in all existing pathogen genomes at each time tick according to a user-specified substitution model. The transition probability matrix $\mathbf{P}$ for nucleotides is a 4×4 matrix where P_{ij} $(i \neq j)$ represents the probability of a nucleotide in state i transitioning to state j in one time tick $(i, j \in \{\text{A, C, G, T}\})$. The overall probability of a site in state i acquiring a mutation in each tick is given by $\sum_{j \neq i} P_{ij}$. If no transition probability matrix is provided, users must specify a mutation rate, defaulting the molecular evolution model to the Jukes–Cantor model (`"subst_model_parameterization": "mut_rate"`).

Otherwise, a mutation probability matrix should be provided (`"subst_model_parameterization": "mut_rate_matrix"`), specifying P_{ij} $(i \neq j)$. Following the format of mutation probability matrix in SLiM, the diagonal values of the mutation probability matrix should be zero (while in practice, the probability of not mutating for allele i is $1 - \sum_{j \neq i} P_{ij}$). To be noted, the provided mutation probability matrix is essentially a transition **probability** matrix in a discrete-time Markov chain, where each time unit is one tick. To simulate the evolution of some specific pathogen, the mutation probability matrix should be scaled to the real time t_0 that each tick represents (unit: real time/tick). For example, a transition rate matrix $\mathbf{Q}$ estimated by popular maximum likelihood inference methods such as RAxML or iqtree, represents instantaneous rate of transition, in the time unit of the time needed for 1 mutation to happen per site. To specify an appropriate transition probability matrix for e3SIM, the molecular clock μ_0 (unit: mutation/site/real time) of this pathogen should be used to calculate the transition probability matrix. The calculated transition probability matrix should be

$$\mathbf{P} = e^{\mathbf{Q}t}$$

where $t = t_0 \times \mu_0$. After changing all the diagonal values to 0s, this matrix could be supplied to e3SIM.

Mutations occur stochastically at a constant probability each tick, as defined by the mutation probability matrix, regardless of whether the pathogen reproduces. Trait values, influenced by the mutation profiles of the pathogens, are recalculated before transmission events. These recalculated trait values subsequently alter the probabilities of relevant transitions, such as transmission and recovery. This mechanism captures the crucial interaction between epi-eco-evo processes, which is often overlooked in classical epidemiological simulators

4.3 Genetic effects to trait values

Causal genomic elements are specific genetic locus in the pathogen genomes that are crucial for the pathogen's functionality and pathogenesis, where mutations significantly impact pathogen traits. In SLiM, the genome is divided into non-overlapping "genomic elements" that are specifed by the starting and ending positions. For each genomic element, one "mutation type" is specified in SLiM. Following this architecture in SLiM, causal genomic elements could be specified in e3SIM. A causal genomic element in e3SIM is specified by its starting and ending position, and any mutation accumulating within these positions confers the same effect size to the relevant trait by an additive architecture.

Currently, two types of pathogen traits can be determined by causal genomic elements: transmissibility and drug resistance. For a pathogen genome i with n mutations in causal genomic elements, the trait value is given by the sum of the effects of all mutations located in the causal genomic elements:

$$\text{Trait}_i = \sum_{j=0}^{n} q_j \times x_{ij}, \tag{4.1}$$

where q_j represents the effect size of mutation j, and x_{ij} is a binary indicator variable that equals 1 if mutation j is present in pathogen genome i and 0 otherwise.

During the execution of the main simulator module, if the causal genomic elements has net positive effect sizes, the trait value is expected to increase as new mutations accumulate at a specified mutation rate. To help users choose a suitable effect size that does not inflate the trait values to an unexpected degree, e3SIM includes an optional heuristic approach for normalizing the expected average trait value to a user-defined value by the end of the simulation by specifying `-normalize T`.

The normalized effect sizes f'_j for mutations in each causal genomic element j for each trait are calculated as:

$$f'_j = \frac{f_j T}{\sum_{j=0}^{n} f_j(\bar{N}_j^{\text{muts}} + \mu g L_j)}. \tag{4.2}$$

Here, f_j represents the unnormalized effect sizes for each mutation in genomic element j. The term $\bar{N}_j^{\text{muts}}$ denotes the average number of mutations in genomic element j across all seeds. L_j denotes the length of genomic element j, and g is the total number of ticks in the simulation, and μ is the mutation probability per site per tick during the simulation. The denominator represents the expected trait value at the final tick of `OutbreakSimulator` before normalization. This normalization provides a baseline that users can manually adjust according to their specific needs.

When a single mutation rate is supplied (`-use_sub_matrix F`), μ is the value supplied by `-mut_rate`. When a mutation probability matrix is supplied (`-use_sub_matrix T`), μ is calculated by assuming the empirical equilibrium of the molecular evolution model. Using X_t to denote the random variable of the state of one site in the pathogen genome at tick t, the probability of mutating per site per tick μ is calculated as:

$$\mu = P(X_{t+1} \neq X_t) = \sum_{x_t \in \{\text{A,C,G,T}\}} \sum_{x_{t+1} \neq x_t} P(X_t = x_t) P(X_{t+1} = x_{t+1} | X_t = x_t)$$

$$= \pi_A(p_{AC} + p_{AC} + p_{AG}) + \pi_C(p_{CA} + p_{CG} + p_{CT}) + \pi_G(p_{GA} + p_{GC} + p_{GT}) + \pi_T(p_{TA} + p_{TC} + p_{TG})$$

where π_k is the equilibrium frequency of state $k \in \{\text{A,C,G,T}\}$, which is calculated empirically as the frequency of each allele in the reference genome provided by `-ref`. p_{uv} is the probability of transitioning from state u to state v in one tick, as provided in `-mu_matrix`.

4.4 Compartmental model

e3SIM employs a classical compartmental epidemiological model by dividing all hosts into four compartments: susceptible (S), exposed (E), infected (I), and recovered (R). Transitions between compartments have base rates that are specified in the configuration file. e3SIM supports a wide range of transitions:

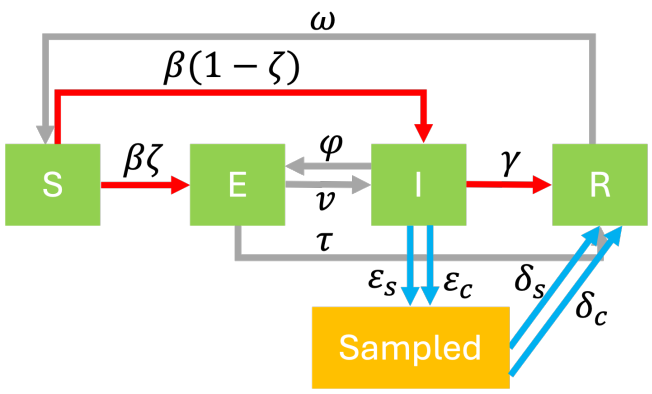


Figure 4.2: **The compartmental model in e3SIM.** Green blocks are the compartments of hosts. The grey arrows show the optional transitions whose probabilities could be specified as a constant, the red arrows are the transition whose probabilities are affected by the relevant trait values of pathogens. The greek letters are the corresponding base transition probabilities, which are fixed within one tick.

Table 4.1: List of symbols for epidemiological parameters. All the probabilities on a per-tick basis.

Symbol	Description	Keys
ω	Probability of immunity loss for a recovered host	`R_S_prob`
β	Base probability of successful infection event for one effective contact	`S_IE_prob`
ζ	Probability of latent infection for each successful infection event	`latency_prob`
ϕ	Probability of deactivation for an infected host	`I_E_prob`
ν	Probability of activation for an exposed host	`E_I_prob`
τ	Probability of transitioning to recovered state for an exposed host	`E_R_prob`
γ	Base probability of clearance for a pathogen residing in an infected host	`I_R_prob`
ϵ_s	Probability of being sampled in a sequential sampling event for an infected host	`sample_prob`
δ_s	Probability of recovery for an infected host following a sequential sampling event	`recovery_prob_after_sampling`
ϵ_c	Probability of being sampled in a concerted sampling event for an infected host	`sampling_prob`
δ_c	Probability of recovery for an infected host following a concerted sampling event	`recovery_prob_after_sampling`

In every tick, **reproduction** happens first. There are two types of reproduction events in e3SIM: transmission and within-host replication.

Transmission occurs when a pathogen infects a new host by producing offspring within the new host, which can only happen when the hosts are in direct contact. During each tick, all effective contacts are evaluated for potential transmissions. An effective contact involves a pair of hosts connected by an edge, where

one host is infected (the infector) and the other is capable of being infected (the infectee). For each effective contact, a single pathogen is randomly selected from the infector's pathogen population. The transmission success of the chosen pathogen is determined by a Bernoulli trial with the success probability defined as the product of the base transmission probability (β) and the pathogen's transmissibility trait:

$$P_i(\text{transmission}) = \beta \times (1 + \text{transmissibility}_i) \tag{4.3}$$

Successful trials result in pathogen transmission to the infectee. e3SIM assumes a transmission bottleneck, where only one pathogen genome from the infector is transmitted to the infectee per successful transmission.

"Super-infection" refers to a scenario in which a host can be infected by multiple sources, either within the same tick or across different ticks. If disabled, a host can only be infected if it is in the susceptible state. In this case, only one successful transmission per tick is retained, with the infecting source randomly chosen from the pool of successful transmissions. If super-infection is enabled, the host can be an infectee regardless of existing infection, as long as the number of pathogens within the host does not exceed the specified maximum capacity. All successful transmissions to this host within a single tick are retained.

Within-host replication occurs when a pathogen reproduces within a single host. This process is enabled only if the within-host replication mechanism is active and the total number of pathogens within the host does not exceed the user-defined maximum capacity. Under these conditions, each pathogen has a probability of producing exactly one offspring within the same host during each simulation tick. Mutations accumulate over time independently of the Section 3.2.2. Therefore, all pathogens, regardless of their reproduction status, have the same probability of mutation per tick.

State changes occur after reproduction events. The transitions for hosts in different states are listed below:

- A newly infected host that is in **susceptible** state before being infected enters the exposed state with a probability ζ and the infected state with a probability $1 - \zeta$. If `superinfection=false`, for hosts that are infected more than once in this tick, only one of the infection events, selected at random, will be kept and all other infection events are omitted. Susceptible hosts that are not involved in transmission events in the current tick remain in the susceptible state.

- For **exposed** hosts, new states are decided by a random draw from a multinoulli distribution with event probabilities v, τ and $1 - \tau - v$, which represent the probability of activation (`E_I_prob`), recovery (`E_R_prob`), and staying exposed.

- For **infected** hosts, new states are decided by a random draw from a multinoulli distribution with event probabilities γ, ϵ, ϕ and $1 - \gamma - \epsilon - \phi$, which represent the probability of recovery evaluation (`I_R_prob`), being sequentially sampled (`sample_prob`), deactivation (`I_E_prob`), and staying infected. For those chosen to be sequentially sampled, an additional Bernoulli trial with parameter δ (`recovery_prob_after_sampling`) determines whether the host ends in the recovered or infected state. If drug resistance is inactive in the current tick, all recovery evaluation yields a successful result so that the hosts transit into the recovered state. However, if drug-resistance is active (`drug-resistance` is not 0 in the current epoch), then for hosts that are evaluated for recovery, each pathogen i within the host undergoes a Bernoulli trial with a successful probability determined by its drug resistance trait:

  $$P_i(\text{survival}) = \text{drug-resistance}_i, \tag{4.4}$$

  If the trial succeeds, the pathogen stays alive during this tick, otherwise, it is cleared. When all pathogens are cleared, the host successfully recovers and transitions to the recovered state, thus the probability of recovery for one host is influenced by the collective resistance traits of all the pathogens in the host. For hosts that are decided to be sequentially sampled, one pathogen is randomly sampled from the host.

- For **recovered** hosts, new states are decided by a Bernoulli trial with probability ω (`R_S_rate`) to see whether they lose their immunity and become susceptible again.

After these sequential state changing events, concerted sampling events happens if specified (specified in the `massive_sampling` section of the configuration file). Concerted sampling only happens in specific ticks, and each event has to be specified by its time-of-happening, sampling probability (ϵ_c) and recovery probability after sampling (δ_c).

Chapter 5

Specifying your own configuration

Contents

In this chapter, examples of different scenarios are provided to help understand how the configuration file and *OutbreakSimulator* works.

5.1 A minimal model for transmissibility

In this section, we will simulate a Tuberculosis outbreak in a host population of size $10,000$ for 10 years. The time scale is that we want each tick to represent 1 day, so we want to simulate 3650 ticks. The mutation rate for tuberculosis genome is 0.5 SNPs per genome per year, which scale to be 3.12×10^{-10} SNPs/bp/day.

1. The first step is to run *NetworkGenerator*. We want to simulate a well-mixed population with every individual having 10 contacts. We thus run:

```
python network_generator.py \
    -popsize 10000 \
    -wkdir ${WKDIR} \
    -method randomly_generate \
    -model ER \
    -p_ER 0.001

```

Listing 5.1: NetworkGenerator model 1

2. We then run *SeedGenerator* to generate the seeds. We want to run a Wright-Fisher model of $N_e = 1000$ and run for 4000 generations. During the burn-in, we want to let one generation scale to one year, and thus use mutation rate 1.1×10^{-7}. We want to generate 5 seeds. The reference genome we use here is Tuberculosis genome, which we put in our repository as well (TB genome).

```
python -u seed_generator.py \
    -wkdir ${WKDIR} \
    -seed_size 5 \
    -method SLiM_burnin_WF \
    -Ne 1000 \
    -ref_path GCF_000195955.2_ASM19595v2_genomic.fna \
    -mu 1.1e-7 \
    -n_gen 4000
```

```
9
```

Listing 5.2: SeedGenerator model 1

If all the seeds coalesce, `seeds.nwk` will appear in the working directory, and we can manually scale the branch lengths by a factor of 365 to scale the tree to based on daily mutation rate. For now, we leave `seeds.nwk` unchanged. It's fine if there isn't a `seeds.nwk` file but we won't be able to generate a full phylogeny tree post *OutbreakSimulator*; rerun this program until all seeds coalesce if desired.

3. Next, we would like to run *GeneticEffectGenerator* using a `.gff` file. A `.gff` file for the TB genome that we preprocessed to ensure no overlapping regions can be found in our repository (TB gff). We also want to normalize the effect sizes based on the timescale of *OutbreakSimulator*. We thus run:

```
    python genetic_effect_generator.py \
        -wkdir ${WKDIR} \
        -method randomly_generate \
        -trait '{"transmissibility": 1, "drug_resistance": 0}' \
        -es_low 1 \
        -es_high 10 \
        -gff GCF_000195955.2_ASM19595v2_genomic.overlap.gff \
        -causal_size_each 5 \
        -normalize T \
        -mut_rate 3.12e-10 \
        -sim_generation 3650

```

Listing 5.3: GeneticEffectGenerator model 1

Running *GeneticEffectGenerator* should output `causal_gene_info.csv` and `seeds_trait_values.csv`. `seeds_trait_values.csv` shows the transmissibility value for each seed based on the genetic architecture that was generated and stored in `causal_gene_info.csv`. You can rerun this program or manually modify `causal_gene_info.csv` and run this program in `-method user_input` mode to tweak the genetic effects.

4. The last step before running *OutbreakSimulator* is matching the seeds to hosts using *HostSeedMatcher*. This time, we want all the seeds to be matched randomly. We can run:

```
    python seed_host_matcher.py \
    -wkdir ${WKDIR} \
    -n_seed 5 \
    -method randomly_generate

```

Listing 5.4: HostSeedMatcher model 1

`seed_host_match.csv` should appear in the working directory.

5. Now, we can run *OutbreakSimulator*. We first download the template configuration file from the Github repository (short template), then make the desired modifications. According to the CDC, latency period for TB can be around 3 months, and recovery time can be 4 months for a rifapentine-moxifloxacin treatment regimen. We want to model all new infections initially leading to latency. Though it's controversial, several studies indicate that immunity doesn't exist in TB. We thus assume that immunity loss occurs at around 20 days. For base transmission probability, we assume an arbitrary probability of 0.004, meaning that for each contact pair, the per-day infection probability is 0.4%. We don't want to model other transitions between compartments, and we don't want to model massive sampling events. This is the configuration file we specify using these parameters:

```
{
  "BasicRunConfiguration": {
    "cwdir": "test_minimal_model",
    "n_replicates": 3
  },
  "EvolutionModel": {
    "n_generation": 3650,
    "mut_rate": 3.12e-10,
    "trans_type": "additive",
    "dr_type": "additive",
    "within_host_reproduction": false,
    "within_host_reproduction_prob": 0,
    "cap_withinhost": 1
  },
  "SeedsConfiguration": {
    "seed_size": 5,
    "use_reference": false
  },
  "GenomeElement": {
    "use_genetic_model": true,
    "ref_path": "GCF_000195955.2_ASM19595v2_genomic.fna",
    "traits_num": '{"transmissibility": 1, "drug_resistance": 0}'
  },
  "NetworkModelParameters": {
    "use_network_model": true,
    "host_size": 10000
  },
  "EpidemiologyModel": {
    "model": "SEIR",
    "epoch_changing": {
      "n_epoch": 1,
      "epoch_changing_generation": []
    },
    "genetic_architecture": {
      "transmissibility": [1],
      "cap_transmissibility": [10],
      "drug_resistance": [0],
      "cap_drugresist": [0]
    },
    "transition_prob": {
      "S_IE_prob": [0.004],
      "I_R_prob: [0.008],
      "R_S_prob": [0.05],
      "latency_prob": [1],
      "E_I_prob": [0.01],
      "I_E_prob": [0],
      "E_R_prob": [0],
      "sample_prob": [0.00001],
      "recovery_prob_after_sampling": [0]
    },
    "massive_sampling": {
      "event_num": 0,
      "generation": [],
      "sampling_prob": [],
      "recovery_prob_after_sampling": []
    },
    "super_infection": false
  },
  "Postprocessing_options": {
    "do_postprocess": true,
```

```
    "tree_plotting": {
      "branch_color_trait": 1,
      "heatmap": "drug_resistance"
    },
    "sequence_output": {
      "vcf": true,
      "fasta": true
    }
  }
}

```

Listing 5.5: config model 1

We thus use the absolute path to this configuration file to run *OutbreakSimulator* by:

```
    python -u outbreak_simulator.py \
        -config ${PATH_TO_THIS_CONFIG}

```

Listing 5.6: OutbreakSimulator model 1

OutbreakSimulator will print logging information to standard output as it progresses through the simulation. When the program finishes, you can look in the working directory and each replicate's subdirectory for the results. The results of one test run are in our repository for comparison (minimal model).

6. For detailed instructions on how to analyze the output files, please refer to Chapter 6.

5.2 Multi-stage drug treatment

In this section, we will simulate a COVID-19 outbreak in a host population of size 10,000 for 2 years. We want each tick to represent 1 day, so we want to simulate 730 ticks. The mutation rate of coronavirus is 6.67×10^{-3} SNPs/bp/year, which is 1.8×10^{-5} SNPs/bp/day. In this simulation, we model realistic host population structure and different treatment stages.

1. The first step is to run *NetworkGenerator*. We want to simulate a realistic host population structure using a scale-free graph. We thus run:

```
    python -u network_generator.py \
        -popsize 10000 \
        -wkdir ${WKDIR} \
        -method randomly_generate \
        -model BA \
        -m 2

```

Listing 5.7: NetworkGenerator model 2

2. In this example, we are starting from the reference genome, thus we dont need to run *SeedGenerator*.

3. We then run *GeneticEffectGenerator*. In this example, we are using a pre-defined effect size file (the provided effect size file).

```
    python -u genetic_effect_generator.py \
        -wkdir ${WKDIR} \
        -method user_input \
        -effsize_path ${EFFSIZE_PATH}$ \
```

```
        -trait_n '{"transmissibility": 1, "drug_resistance": 2}' \
        -n_seed 1
```

Listing 5.8: GeneticEffectGenerator model 2

Here `-trait_n` tells e3SIM the provided effect size file contains one transmissibility and two drug resistance trait sets. You should see `causal_gene_info.csv` appearing in your working directory, along with a `seeds_trait_values.csv` file that only has a header, since we don't have any seed sequences available right now.

4. We then run *HostSeedMatcher*. In this example, we are using only one seed, and we want to match it to a host that has median contact degree. We thus utilize the ranking method to match seed and host.

```
    python -u seed_host_matcher.py \
        -wkdir ${WKDIR} \
        -method randomly_generate \
        -n_seed 1 \
        -match_scheme '{"0": "ranking"}' \
        -match_scheme_param '{"0": 5000}'
```

Listing 5.9: SeedHostMatcher model 2

You should see `seed_host_match.csv` in your working directory.

5. Now, we can run *OutbreakSimulator*. We first download the template configuration file from our Github repository (short template), then make the desired modifications. According to the CDC, latency period for COVID is around 3 days, and recovery time is around 2 weeks. We want to model all new infections initially leading to latency. Though it's controversial, but immunity can lose in 30 days. For basic transmission probability, we assume an arbitrary probability of 0.03, meaning that for each contact pair, an everyday infection probability is 3%. We don't want to model other transitions between compartments, and we don't want to model massive sampling events. We conduct post-processing, asking *OutbreakSimulator* to plot genealogies of the sampled pathogens and include a drug resistance heatmap. This is the configuration file we specify using these parameters:

```
{
  "BasicRunConfiguration": {
    "cwdir": "test_drugresist",
    "n_replicates": 3
  },
  "EvolutionModel": {
    "n_generation": 730,
    "mut_rate": 1.8e-6,
    "trans_type": "additive",
    "dr_type": "additive",
    "within_host_reproduction": false,
    "within_host_reproduction_prob": 0,
    "cap_withinhost": 1
  },
  "SeedsConfiguration": {
    "seed_size": 1,
    "use_reference": true
  },
  "GenomeElement": {
    "use_genetic_model": true,
    "ref_path": "EPI_ISL_402124.fasta",
    "traits_num": '{"transmissibility": 1, "drug_resistance": 2}'
```

```
  },
  "NetworkModelParameters": {
    "use_network_model": true,
    "host_size": 10000
  },
  "EpidemiologyModel": {
    "model": "SEIR",
    "epoch_changing": {
      "n_epoch": 3,
      "epoch_changing_generation": [250, 500]
    },
    "genetic_architecture": {
      "transmissibility": [1, 1, 1],
      "cap_transmissibility": [10, 10, 10],
      "drug_resistance": [0, 1, 2],
      "cap_drugresist": [0, 10, 10]
    },
    "transition_prob": {
      "S_IE_prob": [0.03, 0.03, 0.03],
      "I_R_prob": [0.03, 0.06, 0.05],
      "R_S_prob": [0.05, 0.1, 0.1],
      "latency_prob": [1, 1, 1],
      "E_I_prob": [0.3, 0.3, 0.3],
      "I_E_prob": [0, 0, 0],
      "E_R_prob": [0, 0, 0],
      "sample_prob": [0.0001, 0.0001, 0.0001],
      "recovery_prob_after_sampling": [0, 0, 0]
    },
    "massive_sampling": {
      "event_num": 0,
      "generation": [],
      "sampling_prob": [],
      "recovery_prob_after_sampling": []
    },
    "super_infection": false
  },
  "Postprocessing_options": {
    "do_postprocess": true,
    "tree_plotting": {
      "branch_color_trait": 1,
      "heatmap": "drug_resistance"
    },
    "sequence_output": {
      "vcf": true,
      "fasta": true
    }
  }
}

```

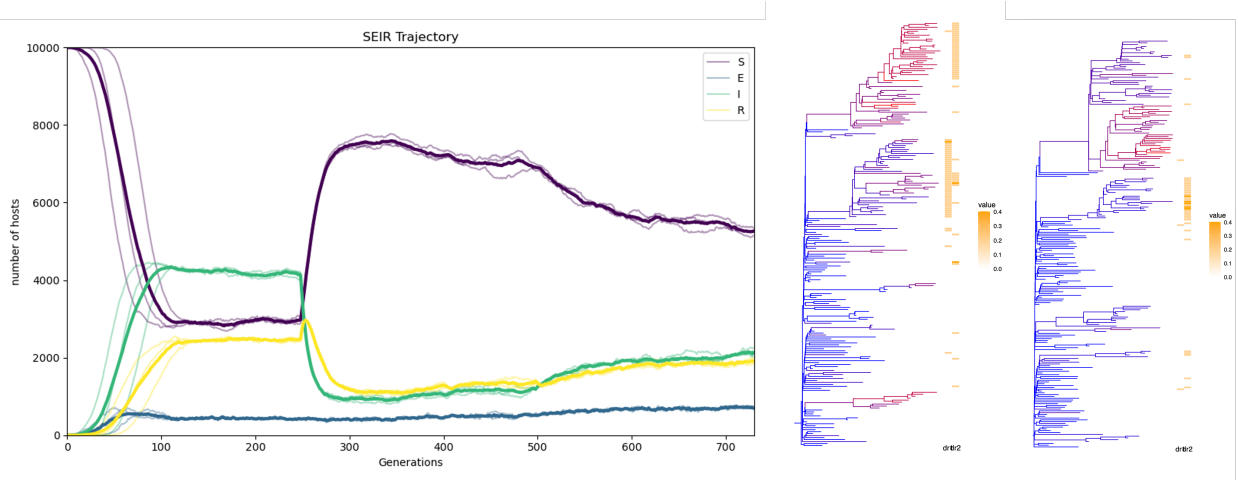
Listing 5.10: config model 2

OutbreakSimulator will print logging information to standard output as it progresses through the simulation. Since this is a stochastic simulation, it isn't guaranteed that the outbreak will reach an endemic stage, i.e. the outbreak might end quickly enough that there aren't many samples retrieved. If no samples are retrieved at all, *OutbreakSimulator* will give a warning and there will be no post-simulation processing steps.

6. When the program finishes, you can look in the working directory and each replicate's subdirectory

for the results. For detailed instructions on how to analyze the output files, please refer to Chapter 6.

7. Below, we show the aggregated SEIR trajectory for three replicates from one simulation run. The different behavior of the outbreak in different epochs can be clearly seen. We also show examples of the transmission trees for two replicates; in both replicates, the sublineages that acquire drug-resistance remain relatively more active in the epochs where drug-resistance is activated. They also accumulate transmissibility-conferring mutations over time.



Part III

Post-simulation processing

Chapter 6

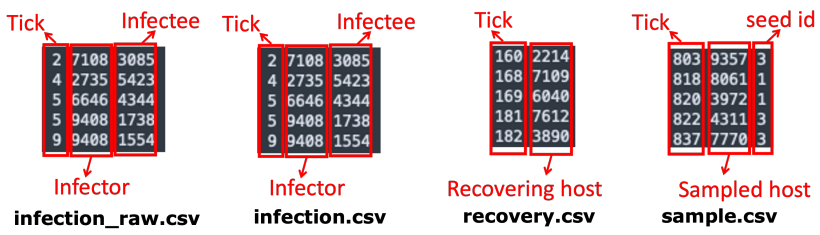
Output

In this chapter, we describe the structure of the output of e3SIM and how to read the outputs.

6.1 Output structure

For every replicate, e3SIM creates a folder named the replicate id (starting from 0) in the working directory. In each replicate folder, there will be the following output files:

- `infection_raw.csz.gz`: Compressed csv file that stores all the raw transmission events (including those events that were omitted when super-infection is deactivated, which do not exist in `infection.csz.gz`). Every row in the file describes one infection event. The first column gives the tick number, and the second and third columns gives the infector and infectee's host ids respectively.
- `infection.csz.gz`: Compressed csv file that stores all the preserved transmission events. For example, when super-infection is deactivated, only one of the transmission events for an infectee is preserved per tick, all the other transmissions were unsuccessful, thus not stored in `infection.csz.gz`. Every row in the file describes one infection event. The first column gives the tick number, and the second and third columns gives the infector and infectee's host ids respectively.
- `recovery.csz.gz`: Compressed csv file that stores all the recovery events that happened. Every row in the file describes one recovery event. The first column gives the tick number and the second column gives the recovering host id.
- `sample.csz.gz`: Compressed csv file that stores all the sampling events that happened. Every row in the file describes one recovery event. The first column gives the tick number and the second column gives the sampled host id. The third column

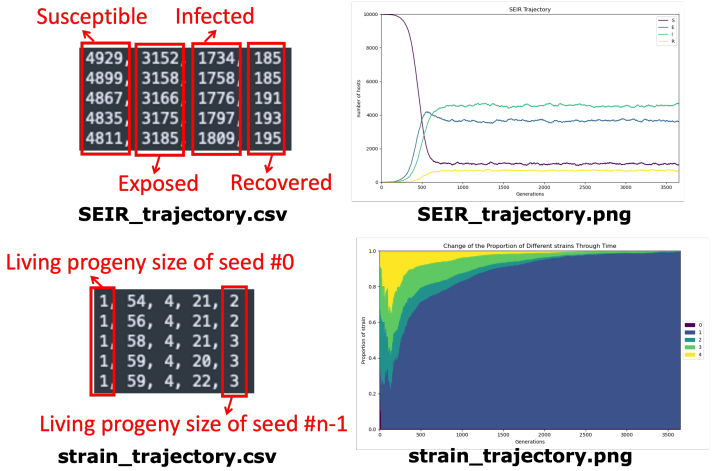


- `SEIR_trajectory.csv.gz`: Compressed csv file that stores the SEIR compartmental size for each tick. It has the row number equals to the total number of ticks. The four columns represents the size of susceptible, exposed, infected and recovered respectively.

- `strain_trajectory.csv.gz`: Compressed csv file that stores the size of the progeny of each strain for each tick. It has the row number equals to the total number of ticks. Each column represents size of the progeny of one seed (strain), in the order of seeds' ids.

- `SEIR_trajectory.png`: Plotted `SEIR_trajectory.csv.gz`.

- `strain_trajectory.png`: Plotted `strain_trajectory.csv.gz` in proportion.

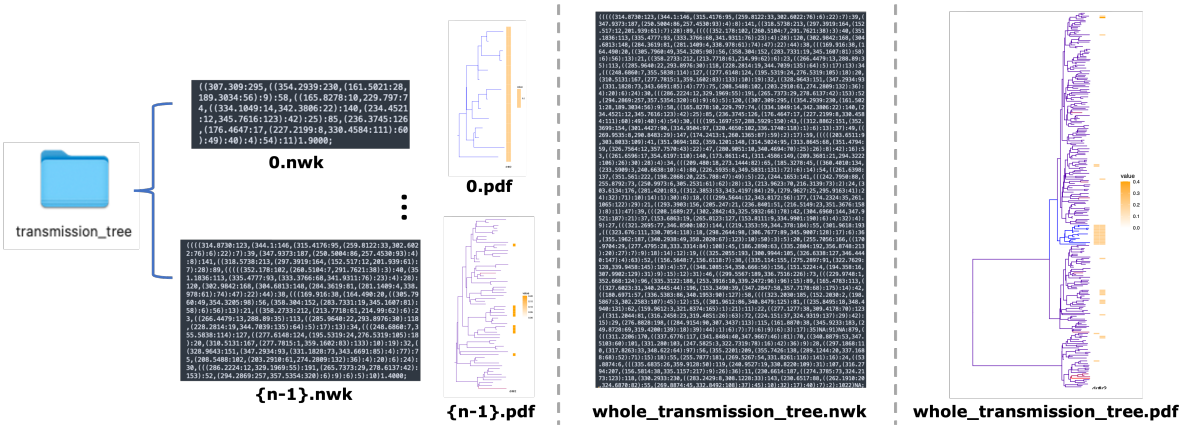


- `transmission_tree`: A folder that stores transmission tree for each seeds' progeny, within which `x.nwk` for each seed id x will present. For seeds that has more than one sampled offspring, a `tree.x.pdf` that is a plotted tree will be present.

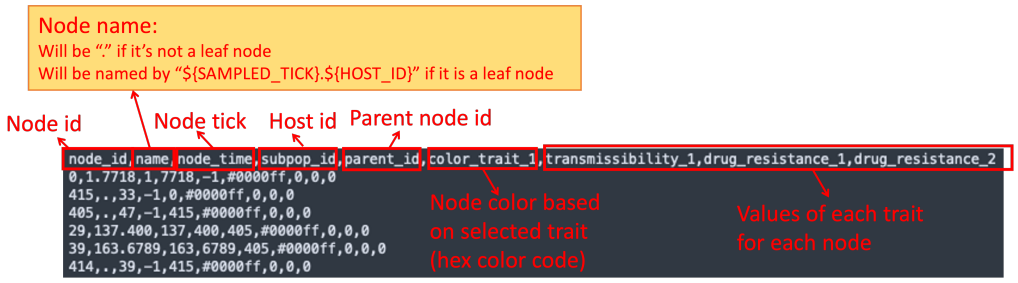
  The leaf labels in the `x.nwk` file is in the format of `${SAMPLED_TICK}.${SAMPLED_HOST_ID}`, which corresponds to the sample names in the `.vcf file`

- `whole_transmission_tree.pdf`: Plotted transmission tree, concatenating the seeds' phylogeny and each seed's transmission tree. Presents only one seed phylogeny presents in the working directory.

- `whole_transmission_tree.nwk`: Transmission tree, concatenating the seeds' phylogeny and each seed's transmission tree. Presents only one seed phylogeny presents in the working directory.



- `sampled_genomes.trees`: Treesequence of the sampled genomes, output by SLiM in `OutbreakSimulator`.

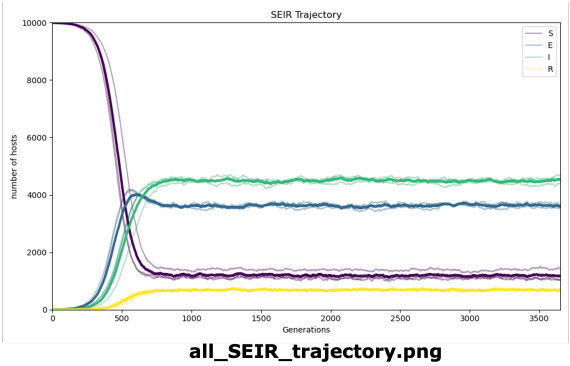


transmission_tree_metadata.nwk

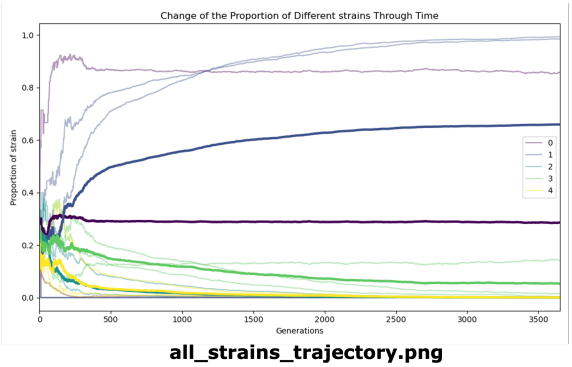
- `transmission_tree_metadata.pdf`: Metadata of the transmission tree, storing the trait values for each node in the transmission tree and the coloring of each node.
- `sampled_pathogen_sequences.vcf`: The mutation profiles of the sampled pathogen sequences in `VCF` format, where sample names correspond to the naming criteria in `whole_transmission_tree.nwk`.
- `sample.SNPs_only.fasta`: A `FASTA` file showing the concatenated SNPs for each sampled pathogen, where sample names correspond to the naming criteria in `whole_transmission_tree.nwk`.
- `final_samples_snp_pos.csv`: A `CSV` file showing the positions of each site corresponding to the reference genome in the concatenated SNPs for each sampled pathogen.
- `sample.wholegenome.fasta`: A `FASTA` file showing the sampled pathogens whole genome sequences, where sample names correspond to the naming criteria in `whole_transmission_tree.nwk`.

In the working directory, there are going to be some outputs as well:

- `all_SEIR_trajectory.png`: Plot the aggregated results from `SEIR_trajectory.csv.gz` for each replicate, with the average trajectory plotted out.
- `all_strain_trajectory.png`: Plot the aggregated results from `strain_trajectory.csv.gz` for each replicate, with the average trajectory plotted out.



all_SEIR_trajectory.png



all_strains_trajectory.png

Part IV

GUI

Chapter 7

GUI

e3SIM provides a graphical user interface for users who prefer visual interactions over command line tools. The GUI could be used for executing all the pre-simulation modules and generating the configuration file for the main simulator module.

To launch the GUI, users should first make sure they are working in the main directory of e3SIM, where they could see a sub-directory named `gui`. Then users could launch the GUI by one command:

```
python gui
```

Listing 7.1: Launch the GUI

A GUI window pops up with several tabs:

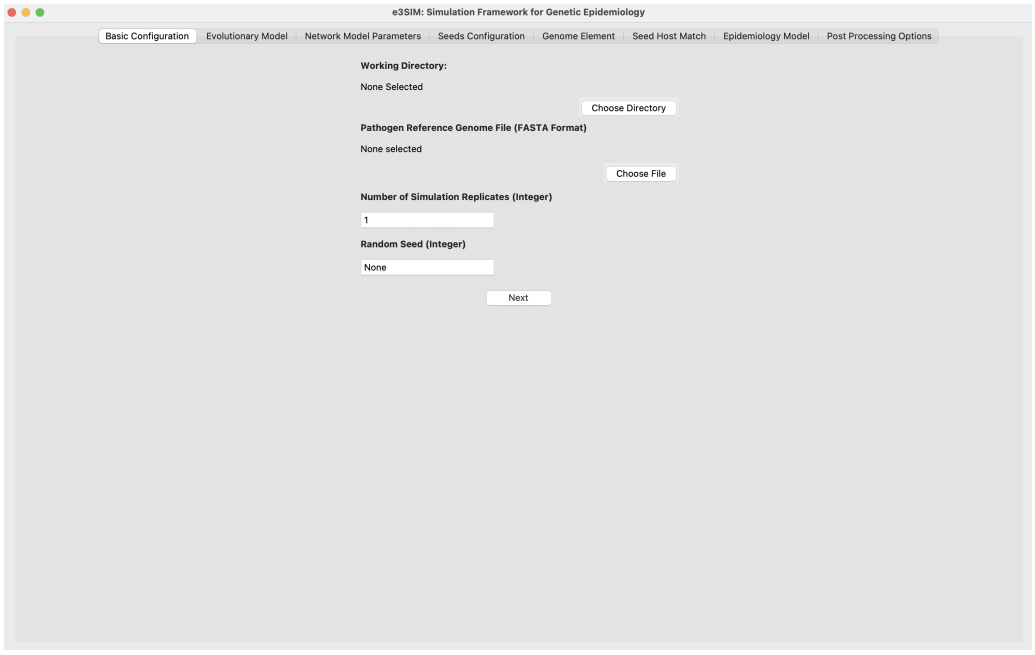


Figure 7.1: **The first tab of the GUI.** In this tab, some basic parameters are set, including working directory and the random number generator, which are used in subsequent tabs where pre-simulation modules are executed.

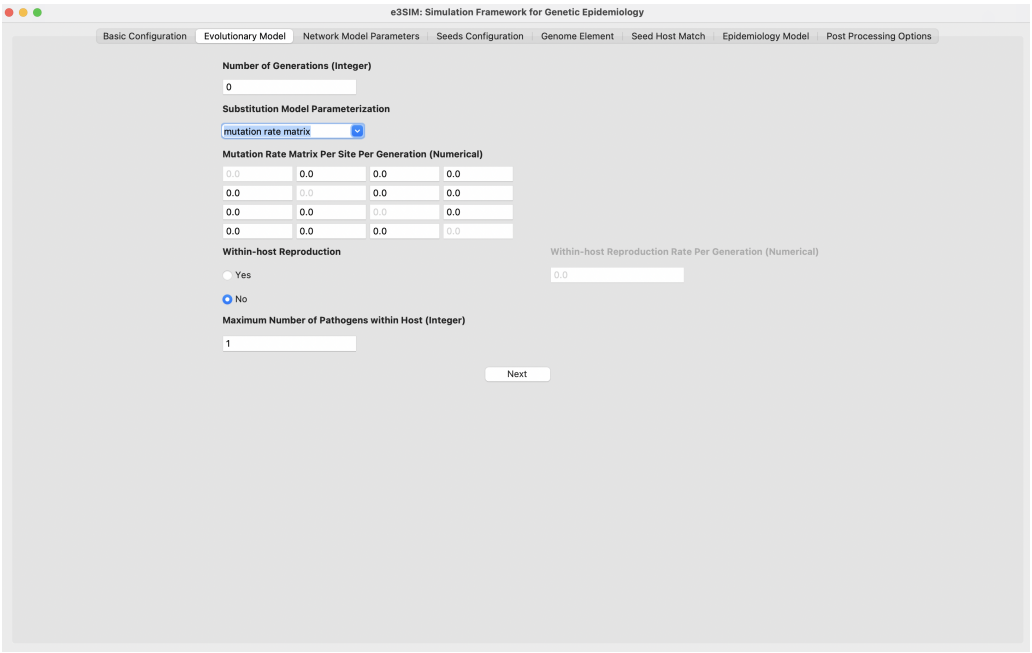


Figure 7.2: **The second tab of the GUI.** In this tab, configurations for the evolutionary model and the within-host reproduction mode are set. This evolutionary model will be used in the main simulator module, and can also be loaded in the *SeedGenerator* module.

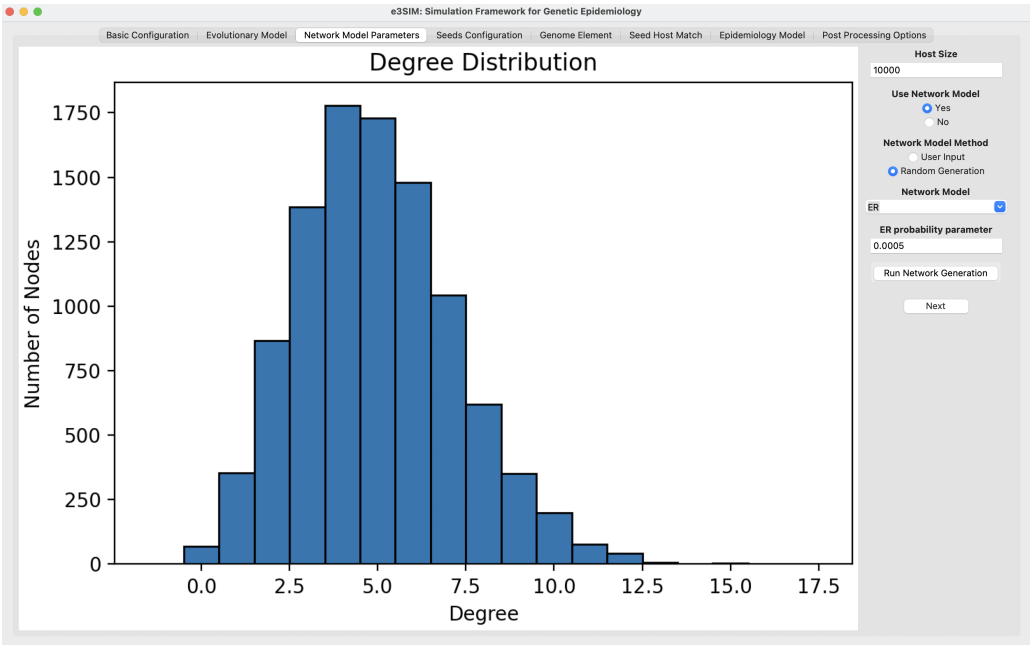


Figure 7.3: **The third tab of the GUI.** In this tab, configurations for the host population are set, and *Network-Generator* is run. The generated contact network will be used in the main simulator and the *HostSeedMatcher* module. After setting all the parameters, clicking on the "Run Network Generation" button will generate a random network and visualize the network in the left panel.

Part V

Advanced usage

Chapter 8

SLiM codes of `outbreakSimulator`

The main simulator module of e3SIM uses SLiM 4 in the back-end. In `outbreakSimulator`, e3SIM assembled Eidos code blocks to generate a customized script for the simulation configuration user specified and executes it. Advanced user could modify these code blocks to implement their own model. All the Eidos code blocks are located in the code directory. For how to write and modify Eidos code and the logic of SLiM, please refer to SLiM's tutorial. In this chapter, we provide the logic of how e3SIM assembles the code blocks, so that experienced users could modify the codes accordingly.

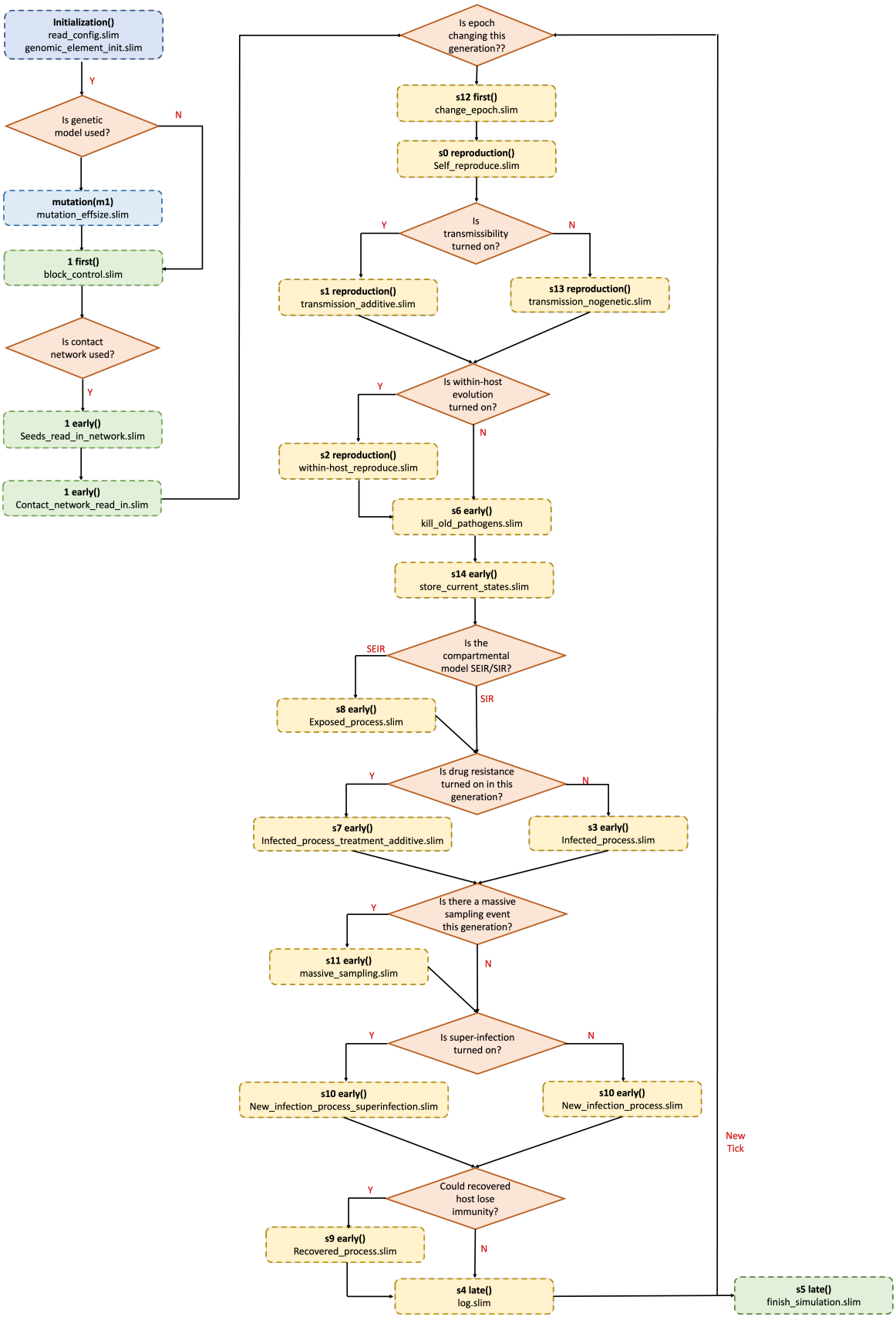


Figure 8.1: Flow chart showing the logic of code block assembling for **OutbreakSimulator**