LASER

Release 1.0.0

Institute for Disease Modeling

Dec 16, 2025

CONTENTS

CHAPTER

ONE

OVERVIEW

docs	
tests	
package	

Light Agent Spatial modeling for ERadication.

- Free software: MIT license

1.1 Installation

```
pip install laser-core
```

You can also install the in-development version with:

```
pip install https://github.com/InstituteforDiseaseModeling/laser/archive/main.zip
```

1.2 Documentation

https://docs.idmod.org/projects/laser/en/latest/

1.3 Development

To run all the tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install laser-core
```

CHAPTER

THREE

USAGE

To use LASER in a project:

```
import laser.core
```

CHAPTER

FOUR

SOFTWARE ARCHITECTURE & DESIGN

4.1 Summary

The LASER framework supports the development of agent-based, spatio-temporal infectious disease models, where the system is represented as a mutable dataframe. Each column corresponds to a numerical property (e.g., `node_id`, `age`, `infection_status`, `infectious_timer`), with rows representing individual agents. The `node_id` property is built-in to accommodate LASER's geospatial focus, though single-node simulations are possible by assigning a uniform `node_id` (e.g., `0` or `10000`). (We may use the term patch interchangeably with node.)

Users define the set of agent properties and implement model behavior through **components**, which consist of an initialization function and a **step function**. Components process one or more properties and run at each timestep to update the 'dataframe'. They can be implemented in **NumPy** (default), **Numba**, or **C** with optimizations like **OpenMP** or **SIMD** for performance. Declarative behavior is encouraged, with step functions optionally described in SQL-like syntax for clarity and maintainability.

The modular architecture enables developers to easily extend the framework by adding custom properties and components, making LASER adaptable to diverse modeling requirements.

4.2 Principles

The core principle of LASER's design is to optimize computational efficiency by aligning the system with what modern CPUs and GPUs excel at—performing billions of floating-point operations per second—while minimizing costly operations like runtime memory allocation and random memory access. To achieve this:

1. **Preallocate Memory:** All required memory (e.g., arrays, data structures) is allocated at initialization. This eliminates the need for dynamic memory allocations during runtime, which can introduce latency and fragmentation.
2. **Sequential Array Access:** Data is processed by iterating sequentially through preallocated arrays, ideally only once per timestep. This design ensures cache-friendly operations and minimizes the overhead of random memory access.
3. **Fixed Data Structures:** Instead of resizing data structures (e.g., appending to lists), the system works with fixed-size arrays where the data for all entities (e.g., agents, reports) is pre-allocated. For instance:
    - Agents are marked as "dead" rather than removed, allowing the array size to remain constant.
    - For births, "preborn" agents are included in the array from the start, with their activation deferred until the appropriate timestep.
4. **Time-Specific Data Slots:** Reports and outputs allocate data slots for every timestep and location in advance. This enables efficient insertion of results during runtime without requiring dynamic resizing.

By adhering to these principles, LASER achieves a highly efficient, scalable system, minimizing the bottlenecks caused by memory management and ensuring smooth timestep progression.

4.3 Layout

The `laser-core` module includes the following core components:

- **LaserFrame**: A custom dataframe class with additional methods tailored for LASER, such as `add_scalar_property` and `add_vector_property`.
- **Demographics Utilities**: Tools to help initialize population demographic data, including dates of birth and expected dates of death.
- **SortedQueue**: A high-performance priority queue class, implemented in the `sortedqueue` submodule, for managing scheduled events like non-disease deaths.
- **PropertySet**: A "smart dictionary" implemented in the `propertyset` submodule that allows dot-notation access to dictionary keys.
- **Migration Module**: A submodule for modeling agent migration using approaches like gravity and radiation models.
- **Visualization Utilities**: Tools for visualizing and analyzing simulation results.

In addition to these core components, users will develop:

- **Top-Level Script**: An application layer, such as `disease.py`, which orchestrates the simulation.
- **Components (Step Functions)**: Modular units that define the dynamics of the simulation, updating one or more agent properties (e.g., age, infection timers) at each timestep.

4.4 Application Layer

The top-level script serves as the orchestrator for the simulation and consists of the following key components:

1. **Configuration Handling**: Gather configuration parameters from the user. This can be achieved via command-line arguments, a `config.json` file, or other input methods. For simple use cases, parameters such as the number of timesteps or the population per node can be hardcoded directly into the script.
2. **Model Initialization**: Create the main Model object, which acts as the container for the simulation's data. This includes the population dataframe (a `LaserFrame` object) and associated metadata or global variables. You can define a detailed Model class or just a minimal stub class that serves as a container.

   Below is an example of the minimal code required to initialize the simplest population dataframe:

```
# Declare a very simple Model class to house our model pieces.
class Model:
    pass

# Initialize the model and its population
from laser.core import LaserFrame
model = Model()
# Create the agent population with max size 1000
model.population = LaserFrame(capacity=1000)
# Add our properties, which can be thought of as the columns of our dataframe.
model.population.add_scalar_property("disease_state")
# Explicitly add the total population size, in this case the same as our max␣
↪capacity
model.population.add(1000)
```

   This code initializes a `LaserFrame` capable of holding 1000 agents, with a single property named `disease_state`. The values for this property will be initialized later, such as through a specific component

or during setup. This represents the simplest functional structure for running a simulation with a basic population and one agent-level property.

3. **Component Setup**: Import and initialize all components (also referred to as phase or step functions). Components define the simulation's logic, such as updating age, managing infections, or handling migration. If all component code resides within the same script, importing may not be necessary.
4. **Simulation Loop**: Execute the main simulation loop. For each timestep, call the step function for every active component in the order defined by your simulation logic. This loop progresses the simulation, updating agent properties and state variables. Counters to track (and record) model state are also updated during each step.
5. **Output and Analysis**: After the simulation completes, generate outputs such as reports, visualizations, or summary statistics. These outputs should provide insights into the simulation's results, such as disease spread, migration patterns, or demographic changes.

4.5 The Model Object

The `Model` object is the central data structure in LASER, encapsulating both agent and node-level information using `LASERFrame`. This structure is a specialized dataframe designed for managing and updating model properties efficiently.

- **Agent Population**: Represented as a `LASERFrame`, where each agent corresponds to a row and each property (e.g., age, infection status) is stored as a column. This allows for efficient per-agent computations during each timestep.
- **Node Data**: Another `LASERFrame` is used to manage node-level information. - **Input Values**: These are typically static scalar properties (e.g., geographic coordinates, demographic constants) provided at initialization. - **Output Values**: These include dynamic vector properties (e.g., total population or infected count at each timestep) that are updated as the simulation progresses.

As the `Model` evolves, additional data and methods may be incorporated into the class to better support the specific needs of your simulation. This flexibility ensures the `Model` can serve as a robust container for all simulation-related data and computations.

4.6 Components

Components are modular units of functionality within the simulation, responsible for performing specific updates or computations on the agent population or node-level data. Each component is implemented as a class with an initialization function to set up any required state and a step function to execute the component's logic during each timestep.

As demonstrated in the "Model Initialization" section, the `LaserFrame` object contains the agent properties. Components operate on these properties to modify their values or derive new ones based on simulation logic.

4.6.1 Example: Infection Timer Component

The example below shows a component that decrements the `infection_timers` property for all agents. When a timer reaches zero, the component sets the corresponding agent's `susceptibility` flag to reflect that they have recovered or gained immunity.

```
class InfectionTimerComponent:
    def __init__(self, model):
        self.population = model.population

    def step(self):
        timers = self.population.infection_timers
        susceptibility = self.population.susceptibility
```

(continues on next page)

(continued from previous page)

```
        # Decrement all non-zero timers
        timers[:] = np.maximum(timers - 1, 0)

        # Update susceptibility based on timer state
        susceptibility[:] = np.where(timers == 0, 1, susceptibility)
```

After defining this component, it would typically be initialized and added to the simulation loop:

```
# Initialize the component
infection_timer_component = InfectionTimerComponent(model)

# Run the component step function during the simulation
for timestep in range(total_timesteps):
    infection_timer_component.step()
```

4.6.2 Explanation

1. **Initialization**: The component retrieves a reference to the `LaserFrame` from the model. This allows direct access to the agent properties.
2. **Step Function**: The logic modifies the `infection_timers` array in place, ensuring that values do not go below zero, and updates the `susceptibility` flag based on timer state.
3. **Integration**: The component is called once per timestep, ensuring its behavior aligns with the simulation's temporal dynamics.

By defining components in this modular fashion, the LASER framework supports reusable and extensible functionality, allowing developers to add new behavior to simulations efficiently.

4.7 Input Files

There is no requirement for any particular input files for laser-core. You're free to provide, load and parse input data in preferred formats for values such as input populations, age structure, fertility, mortality, and migration rates.

4.8 Output Files

laser-core does not output data to disk. It's up to you to collect and write csv or other data files as needed. HDF5 file format is preferred for large output files.

4.9 Demographics

4.9.1 Age Structure

If you want to work with age structure for a short simulation which doesn't need births you can just give everyone an age (based on distribution) and increment it each timestep. The laser.core.demographics.pyramid module is provided to support the initialization of agents with plausible initial ages.

4.9.2 Births

Preborn Management in LASER

LASER's design philosophy emphasizes contiguous and fixed-size arrays, meaning all agents—both currently active and preborn—are created at the start of the simulation. Preborns are "activated" as they are born, rather than being dynamically added. Several approaches to handling preborns while adhering to these principles are outlined below:

1. **Negative and Positive Birthdays**: - Assign `date_of_birth` values in the past (negative) for active agents. - Assign `date_of_birth` values in the future (positive) for preborns.
2. **Unified Preborn Marker**: - Set all preborns' `date_of_birth` to a placeholder value (e.g., `-1`). - Update the `date_of_birth` to the current timestep when a preborn is born.
3. **Active Flag Only** (if not modeling age structure): - If the model doesn't require age structure, you can skip `date_of_birth` entirely. Instead, use an `active` flag. Preborns start with `active = False` and are switched to `active = True` during the fertility step. This simplifies implementation while remaining consistent with LASER principles.

Calculating Age from Birthday

If calculating age isn't frequent or essential, you can avoid explicitly tracking an `age` property. Instead, compute age dynamically as the difference between the current timestep (`now`) and `date_of_birth`. For models that depend on age-specific dynamics (e.g., fertility rates by age group), consider adding a dedicated `age` property that updates at each timestep.

4.9.3 Deaths

The recommended way of doing mortality in LASER is by precalculating a lifespan for each agent, rather than probabilistically killing agents as the simulation runs. This can take different forms: If you prefer to track agent age, you can also have an agent lifespan. Alternatively, if you are just using *date_of_birth* you can have a *date_of_death*, where theses 'dates' are really simulation times ('sim day of birth' and 'sim day of death'). Also, in LASER, as mentioned in the 'Principles' section, we strive to leave the contiguous arrays of agent data in place, without adding or deleting elements (allocating or freeing). This means that to model mortality, we prefer to 'kill' agents by doing either 1) check that their age is greater than their lifespan (or that the current timestep is greater than their 'sim day of death') in each component that cares, or 2) Set an active flag to false or a dead flag to true. The second approach is simpler, and avoids doing millions of comparison operations, at the cost of an additional property. Note that many component operations (step functions) can be done without checking whether the agent is alive, because, for example, as long as transmission never infects a dead person, decrementing all non-zero infection timers will only operate on live agents. Finally, while you can set lifespans using any algorith you want, laser.core.demographics.kmestimator is provided to support these calculations.

4.10 User Customizability

1. **Config Params** LASER doesn't have a set of pre-existing configuration params. You are free to add code to let the user set params like R-nought or simulation duration in code, in a settings file, on the command line, or even in environment variables. We suggest you collect these early in the sim and store them in a PropertySet which is then stored as a member of the model.

2. **Input Files** LASER doesn't have a set of pre-defined input files or file formats but it's likely as you develop your model that you will want to load population data (by node/patch) and other demographics from csv files. This can provide a convient data-driven way of modifying model behavior.

3. **Code** As discussed above, LASER modelers are expected to write their own application-level scripts and their own components.

4.11 New Modeler Workflow

Here's how you should break down your modeling problem to model a disease with LASER:

1. Figure out how your disease model maps to a set of agent properties.
2. Add code to add those properties to the population LASERFrame.
3. Figure out the updates you'll need to do each timestep, as declarations.
4. Add component code for each of those updates.

4.12 Glossary of Terms

- **Patch** Something…

CHAPTER

FIVE

LASER.CORE

5.1 laser.core package

5.1.1 Subpackages

laser.core.demographics package

Submodules

laser.core.demographics.kmestimator module

This module provides the KaplanMeierEstimator class for predicting the year and age at death based on given ages and cumulative death data.

Classes:

- KaplanMeierEstimator: A class to perform Kaplan-Meier estimation for predicting the year and age at death.

Functions:

- _pyod(ages_years: np.ndarray, cumulative_deaths: np.ndarray, max_year: np.uint32 = 100): Calculate the predicted year of death based on the given ages in years.
- _pdod(age_in_days: np.ndarray, year_of_death: np.ndarray, day_of_death: np.ndarray): Calculate the predicted day of death based on the given ages in days and predicted years of death.

Usage example:

```
` estimator = KaplanMeierEstimator(cumulative_deaths=np.array([...])) year_of_death
= estimator.predict_year_of_death(np.array([40, 50, 60]), max_year=80) age_at_death
= estimator.predict_age_at_death(np.array([40*365, 50*365, 60*365]), max_year=80) `
```

class laser.core.demographics.kmestimator.**KaplanMeierEstimator**(*source: ndarray | list | Path | str*)

Bases: `object`

property cumulative_deaths: ndarray

Returns the original source data.

predict_age_at_death(*ages_days: ndarray[Any, dtype[integer]]*, *max_year: uint32 = None*) → ndarray

Calculate the predicted age at death (in days) based on the given ages in days.

Parameters

- **ages_days** (*np.ndarray*) – The ages of the individuals in days.
- **max_year** (*int, optional*) – The maximum year to consider for calculating the predicted year of death. Default is None, which uses the maximum year from the source data.

Returns
age_at_death (np.ndarray) – The predicted days of death.

Example

` predict_age_at_death(np.array([40*365, 50*365, 60*365]), max_year=80) # returns something like array([22732, 26297, 29862]) `

predict_year_of_death(*ages_years: ndarray[Any, dtype[integer]]*, *max_year: uint32 = None*) → ndarray

Calculate the predicted year of death based on the given ages in years.

Parameters

- **ages_years** (*np.ndarray*) – The ages of the individuals in years.
- **max_year** (*int, optional*) – The maximum year to consider for calculating the predicted year of death. Default is None, which uses the maximum year from the source data.

Returns
year_of_death (np.ndarray) – The predicted years of death.

Example

` predict_year_of_death(np.array([40, 50, 60]), max_year=80) # returns something like array([62, 72, 82]) `

sample(*current: ndarray[Any, dtype[integer]]*, *max_index: uint32 = None*) → ndarray

Similar to *predict_year_of_death*, but operates on indices rather than years. This method predicts the expiration (death) index for each individual, given their current index.

Parameters

- **current** (*np.ndarray*) – The current indices of the individuals.
- **max_index** (*int, optional*) – The maximum index to consider for calculating the predicted expiration. Default is None, which uses the maximum index from the source data.

Returns
predictions (np.ndarray) – The predicted expiration indices for each individual.

laser.core.demographics.pyramid module

A class for generating samples from a distribution using the Vose alias method.

class laser.core.demographics.pyramid.**AliasedDistribution**(*counts*)

Bases: object

A class to generate samples from a distribution using the Vose alias method.

property alias: ndarray

property probs: ndarray

sample(*count=1*, *dtype=<class 'numpy.int32'>*) → int

Generate samples from the distribution.

Parameters
count (*int*) – The number of samples to generate. Default is 1.

Returns
int or numpy.ndarray – A single integer if count is 1, otherwise an array of integers representing the generated samples.

property total: int

laser.core.demographics.pyramid.**load_pyramid_csv**(*file: Path*, *verbose=False*) → ndarray

Load a CSV file with population pyramid data and return it as a NumPy array.

The CSV file is expected to have the following schema:

- The first line is a header: "Age,M,F"
- Subsequent lines contain age ranges and population counts for males and females:

` "low-high,#males,#females" ... "max+,#males,#females" ` Where low, high, males, females, and max are integer values >= 0.

The function processes the CSV file to create a NumPy array with the following columns:

- Start age of the range
- End age of the range
- Number of males
- Number of females

Parameters

- **file** (*Path*) – The path to the CSV file.
- **verbose** (*bool*) – If True, prints the file reading status. Default is False.

Returns

np.ndarray – A NumPy array with the processed population pyramid data.

laser.core.demographics.spatialpops module

laser.core.demographics.spatialpops.**distribute_population_skewed**(*tot_pop*, *num_nodes*, *frac_rural=0.3*)

Calculate the population distribution across a number of nodes based on a total population, the number of nodes, and the fraction of the population assigned to rural nodes.

The function generates a list of node populations distributed according to a simple exponential random distribution, with adjustments to ensure the sum matches the total population and the specified fraction of rural population is respected.

Parameters

- **tot_pop** (*int*) – The total population to be distributed across the nodes.
- **num_nodes** (*int*) – The total number of nodes among which the population will be distributed.
- **frac_rural** (*float*) – The fraction of the total population to be assigned to rural nodes (value between 0 and 1). Defaults to 0.3. The 0 node is the single urban node and has (1 - frac_rural) of the population.

Returns

(list[int]) – A list of integers representing the population at each node. The sum of the list equals *tot_pop*.

Notes

- The population distribution is weighted using an exponential random distribution to create heterogeneity among node populations.
- Adjustments are made to ensure the total fraction assigned to rural nodes adheres to *frac_rural*.

Examples

```
>>> from laser.core.demographics.spatialpops import distribute_population_skewed
>>> np.random.seed(42)  # For reproducibility
>>> tot_pop = 1000
>>> num_nodes = 5
>>> frac_rural = 0.3
>>> distribute_population_skewed(tot_pop, num_nodes, frac_rural)
[700, 154, 64, 54, 28]
```

```
>>> tot_pop = 500
>>> num_nodes = 3
>>> frac_rural = 0.4
>>> distribute_population_skewed(tot_pop, num_nodes, frac_rural)
[300, 136, 64]
```

laser.core.demographics.spatialpops.**distribute_population_tapered**(*tot_pop*, *num_nodes*)

Distribute a total population heterogeneously across a given number of nodes.

The distribution follows a logarithmic-like decay pattern where the first node (Node 0) receives the largest share of the population, approximately half the total population. Subsequent nodes receive progressively smaller populations, ensuring that even the smallest node has a non-negligible share.

The function ensures the sum of the distributed populations matches the *tot_pop* exactly by adjusting the largest node if rounding introduces discrepancies.

Parameters

- **tot_pop** (*int*) – The total population to distribute. Must be a positive integer.
- **num_nodes** (*int*) – The number of nodes to distribute the population across. Must be a positive integer.

Returns

ndarray (numpy.ndarray) –

A 1D array of integers where each element represents the population assigned to a specific node. The length of the array is equal to *num_nodes*.

Raises

ValueError – If *tot_pop* or *num_nodes* is not greater than 0.

Notes

- The logarithmic-like distribution ensures that Node 0 has the highest population, and subsequent nodes receive progressively smaller proportions.
- The function guarantees that the sum of the returned array equals *tot_pop*.

Examples

Distribute a total population of 1000 across 5 nodes:

```
>>> from laser.core.demographics.spatialpops import distribute_population_tapered
>>> distribute_population_tapered(1000, 5)
array([500, 250, 125, 75, 50])
```

Distribute a total population of 1200 across 3 nodes:

```
>>> distribute_population_tapered(1200, 3)
array([600, 400, 200])
```

Handling a small total population with more nodes:

```
>>> distribute_population_tapered(10, 4)
array([5, 3, 2, 0])
```

Ensuring the distribution adds up to the total population:

```
>>> pop = distribute_population_tapered(1000, 5)
>>> pop.sum()
1000
```

Module contents

class laser.core.demographics.**AliasedDistribution**(*counts*)

Bases: object

A class to generate samples from a distribution using the Vose alias method.

property alias: ndarray

property probs: ndarray

sample(*count=1*, *dtype=<class 'numpy.int32'>*) → int

Generate samples from the distribution.

Parameters

count (*int*) – The number of samples to generate. Default is 1.

Returns

int or numpy.ndarray – A single integer if count is 1, otherwise an array of integers representing the generated samples.

property total: int

class laser.core.demographics.**KaplanMeierEstimator**(*source: ndarray | list | Path | str*)

Bases: object

property cumulative_deaths: ndarray

Returns the original source data.

predict_age_at_death(*ages_days: ndarray[Any, dtype[integer]]*, *max_year: uint32 = None*) → ndarray

Calculate the predicted age at death (in days) based on the given ages in days.

Parameters

- **ages_days** (*np.ndarray*) – The ages of the individuals in days.

- **max_year** (*int, optional*) – The maximum year to consider for calculating the predicted year of death. Default is None, which uses the maximum year from the source data.

Returns
age_at_death (np.ndarray) – The predicted days of death.

Example

` predict_age_at_death(np.array([40*365, 50*365, 60*365]), max_year=80) # returns something like array([22732, 26297, 29862]) `

predict_year_of_death(*ages_years: ndarray[Any, dtype[integer]]*, *max_year: uint32 = None*) → ndarray

Calculate the predicted year of death based on the given ages in years.

Parameters

- **ages_years** (*np.ndarray*) – The ages of the individuals in years.
- **max_year** (*int, optional*) – The maximum year to consider for calculating the predicted year of death. Default is None, which uses the maximum year from the source data.

Returns
year_of_death (np.ndarray) – The predicted years of death.

Example

` predict_year_of_death(np.array([40, 50, 60]), max_year=80) # returns something like array([62, 72, 82]) `

sample(*current: ndarray[Any, dtype[integer]]*, *max_index: uint32 = None*) → ndarray

Similar to *predict_year_of_death*, but operates on indices rather than years. This method predicts the expiration (death) index for each individual, given their current index.

Parameters

- **current** (*np.ndarray*) – The current indices of the individuals.
- **max_index** (*int, optional*) – The maximum index to consider for calculating the predicted expiration. Default is None, which uses the maximum index from the source data.

Returns
predictions (np.ndarray) – The predicted expiration indices for each individual.

laser.core.demographics.**load_pyramid_csv**(*file: Path*, *verbose=False*) → ndarray

Load a CSV file with population pyramid data and return it as a NumPy array.

The CSV file is expected to have the following schema:

- The first line is a header: "Age,M,F"
- Subsequent lines contain age ranges and population counts for males and females:

` "low-high,#males,#females" ... "max+,#males,#females" ` Where low, high, males, females, and max are integer values >= 0.

The function processes the CSV file to create a NumPy array with the following columns:

- Start age of the range
- End age of the range
- Number of males
- Number of females

Parameters

- **file** (*Path*) – The path to the CSV file.
- **verbose** (*bool*) – If True, prints the file reading status. Default is False.

Returns

np.ndarray – A NumPy array with the processed population pyramid data.

5.1.2 Submodules

5.1.3 laser.core.cli module

Module that contains the command line app.

Why does this file exist, and why not put this in __main__?

> You might be tempted to import things from __main__ later, but that will cause problems: the code will get executed twice:
>
> - When you run *python -midmlaser* python will execute `__main__.py` as a script. That means there will not be any `idmlaser.__main__` in `sys.modules`.
> - When you import __main__ it will get executed again (as a module) because there"s no `idmlaser.__main__` in `sys.modules`.
>
> Also see (1) from https://click.palletsprojects.com/en/stable/setuptools/

5.1.4 laser.core.distributions module

Various probability distributions implemented using NumPy and Numba.

LASER based models generally move from pure NumPy to Numba-accelerated version of the core dynamics, e.g., transmission.

It would be a hassle to re-implement these functions for each desired distribution, so we provide these Numba-wrapped distributions here which can be passed in to other Numba compiled functions.

For example, a simple SIR model may want to parameterize the infectious period using a distribution. By passing in a Numba-wrapped distribution function, we can pick and parameterize a distribution based on configuration and sample from that distribution within the Numba-compiled SIR model without needing to re-implement the distribution logic within the SIR model itself.

A simple example of usage:

```
import laser.core.distributions as dist

# Create a Numba-wrapped beta distribution
beta_dist = dist.beta(2.0, 5.0)

# Assign the distribution to the model's infectious period distribution
# so the transmission component can sample from it during simulation
model.infectious_period_dist = beta_dist
```

Note that the distribution functions take two parameters, `tick` and `node`, which are currently unused but match the desired signature for disease model components that may need to sample from distributions based on the current simulation tick or node index. In other words, distributions with spatial or temporal variation could be implemented in the future.

Here are examples of Numba-wrapped distribution functions that could vary based on tick or tick + node:

```
# temporal variation only
cosine = np.cos(np.linspace(0, np.pi * 2, 365))

@nb.njit(nogil=True, cache=True)
def seasonal_distribution(tick: int, node: int) -> np.float32:
    # ignore node for this seasonal distribution
    day_of_year = tick % 365
    base_value = 42.0 + 3.14159 * cosine[day_of_year]
    # parameterize normal() with seasonal factor
    return np.float32(np.random.normal(base_value, 2.0))

# additional spatial variation
ramp = np.linspace(0, 2, 42)

@nb.njit(nogil=True, cache=True)
def ramped_distribution(tick: int, node: int) -> np.float32:
    day_of_year = tick % 365
    # use seasonal factor
    base_value = 42.0 + 3.14159 * cosine[day_of_year]
    # apply spatial ramp based on node index
    base_value *= ramp[node]
    # parameterize normal() with seasonal + spatial factor
    return np.float32(np.random.normal(base_value, 1.0))
```

Normally, these distributions—built in or custom—will be used once per agent as above. However, the `sample_ints()` and `sample_floats()` functions can be used to efficiently sample large arrays using multiple CPU cores in parallel.

`laser.core.distributions.`**`beta`**(*a*, *b*)

Beta distribution.

$f(x; a, b) = frac {x^{a-1} (1-x)^{b-1}} {B(a, b)}$

where $B(a, b)$ is the beta function.

`laser.core.distributions.`**`binomial`**(*n*, *p*)

Binomial distribution.

$f(k,n,p) = Pr(X = k) = binom {n} {k} p^k (1-p)^{n-k}$

where n is the number of trials and p is the probability of success [0, 1].

`laser.core.distributions.`**`constant_float`**(*value*)

Constant distribution. Always returns the same floating point value.

`laser.core.distributions.`**`constant_int`**(*value*)

Constant distribution. Always returns the same integer value.

`laser.core.distributions.`**`exponential`**(*scale*)

Exponential distribution.

$f(x; frac {1} {beta}) = frac {1} {beta} e^{-frac {x} {beta}}$

where $beta$ is the scale parameter ($beta = 1 / lambda$).

`laser.core.distributions.`**`gamma`**(*shape*, *scale*)

Gamma distribution.

$p(x) = x^{k-1} frac {e^{- x / theta}}{theta^k Gamma(k)}$

where k is the shape, $theta$ is the scale, and $Gamma(k)$ is the gamma function.

laser.core.distributions.**logistic**(*loc*, *scale*)

Logistic distribution.

$P(x) = frac {e^{-(x - mu) / s}} {s (1 + e^{-(x - mu) / s})^2}$

where mu is the location parameter and s is the scale parameter.

laser.core.distributions.**lognormal**(*mean*, *sigma*)

Log-normal distribution.

$p(x) = frac {1} {sigma x sqrt {2 pi}} e^{- frac {(ln x - mu)^2} {2 sigma^2}}$

where mu is the mean and $sigma$ is the standard deviation of the underlying normal distribution.

laser.core.distributions.**negative_binomial**(*n*, *p*)

Negative binomial distribution.

$P(N; n, p) = frac {Gamma (N + n)} {N! Gamma (n)} p^n (1 - p)^N$

where n is the number of successes, p is the probability of success on each trial, $N + n$ is the number of trials, and $Gamma()$ is the gamma function. When n is an integer,

$frac {Gamma (N + n)} {N! Gamma (n)} = binom {N + n - 1} {n - 1}$

which is the more common form of this term.

laser.core.distributions.**normal**(*loc*, *scale*)

Normal (Gaussian) distribution.

$p(x) = frac {1} {sqrt {2 pi sigma^2}} e^{- frac {(x - mu)^2} {2 sigma^2}}$

where mu is the mean and $sigma$ is the standard deviation.

laser.core.distributions.**poisson**(*lam*)

Poisson distribution.

$f(k ; lambda) = frac {lambda^k e^{- lambda}} {k!}$

where $lambda$ is the expected number of events in the given interval.

laser.core.distributions.**sample_floats**(*fn*, *dest*, *tick=0*, *node=0*)

Fill an array with floating point values sampled from a Numba-wrapped distribution function.

Parameters

- **fn** (*function*) – Numba-wrapped distribution function returning float32 values.
- **dest** (*np.ndarray*) – Pre-allocated destination float32 array to store samples.
- **tick** (*int, optional*) – Current simulation tick (default is 0). Passed through to the distribution function.
- **node** (*int, optional*) – Current node index (default is 0). Passed through to the distribution function.

Returns

dest (np.ndarray) – The destination array filled with sampled values.

laser.core.distributions.**sample_ints**(*fn*, *dest*, *tick=0*, *node=0*)

Fill an array with integer values sampled from a Numba-wrapped distribution function.

Parameters

- **fn** (*function*) – Numba-wrapped distribution function returning int32 values.

- **dest** (*np.ndarray*) – Pre-allocated destination int32 array to store samples.
- **tick** (*int, optional*) – Current simulation tick (default is 0). Passed through to the distribution function.
- **node** (*int, optional*) – Current node index (default is 0). Passed through to the distribution function.

Returns
dest (np.ndarray) – The destination array filled with sampled values.

`laser.core.distributions.`**`uniform`**(*low*, *high*)

Uniform distribution.

$p(x) = frac {1} {b - a}$

where a is the lower bound and b is the upper bound, [a, b).

`laser.core.distributions.`**`weibull`**(*a*, *lam*)

Weibull distribution.

$X = lambda (- ln (U))^{1 / a}$

where a is the shape parameter and $lambda$ is the scale parameter.

5.1.5 laser.core.extension module

`laser.core.extension.`**`compiled`**(*args*)

5.1.6 laser.core.laserframe module

laserframe.py

This module defines the LaserFrame class, which is used to manage dynamically allocated data for agents or nodes/patches. The LaserFrame class is similar to a database table or a Pandas DataFrame and supports scalar and vector properties.

Classes:
LaserFrame: A class to manage dynamically allocated data for agents or nodes/patches.

Usage Example:
```
`python laser_frame = LaserFrame(capacity=100) laser_frame.add_scalar_property('age',
dtype=np.int32, default=0) laser_frame.add_vector_property('position', length=3,
dtype=np.float32, default=0.0) start, end = laser_frame.add(10) laser_frame.sort(np.
arange(10)[::-1]) laser_frame.squash(np.array([True, False, True, False, True,
False, True, False, True, False])) `
```

ivar count
The current count of active elements.

vartype count
int

ivar capacity
The maximum capacity of the frame.

vartype capacity
int

Note

Since count can be less than capacity, properties return slices of the underlying arrays up to count by default so users do not have to include the slice themselves. I.e., if *lf* is a LaserFrame, then *lf.age* returns *lf._age[0:lf.count]* automatically. The full underlying array is always available as *lf._age* (or whatever the property name is). The slice returned is valid for all NumPy operations, including assignment, as well as for use with Numba compiled functions.

class laser.core.laserframe.**LaserFrame**(*capacity: int*, *initial_count: int = -1*, ***kwargs*)

Bases: object

The LaserFrame class, similar to a db table or a Pandas DataFrame, holds dynamically allocated data for agents (generally 1-D or scalar) or for nodes|patches (e.g., 1-D for scalar value per patch or 2-D for time-varying per patch).

add(*count: int*) → tuple[int, int]

Adds the specified count to the current count of the LaserFrame.

This method increments the internal count by the given count, ensuring that the total does not exceed the frame's capacity. If the addition would exceed the capacity, an assertion error is raised. This method is typically used to add new births during the simulation.

Parameters
count (*int*) – The number to add to the current count.

Returns
tuple[int, int] – A tuple containing the [start index, end index) after the addition.

Raises
AssertionError – If the resulting count exceeds the frame's capacity.

add_array_property(*name: str*, *shape: tuple*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Adds an array property to the object.

This method initializes a new property with the given name as a multi-dimensional NumPy array.

The array will have the given shape (note that there is no implied dimension of size self._capacity), datatype (default is np.uint32), and default value (default is 0).

Parameters

- **name** (*str*) – The name of the property to be added.
- **shape** (*tuple*) – The shape of the array.
- **dtype** (*data-type, optional*) – The desired data-type for the array, default is np.uint32.
- **default** (*scalar, optional*) – The default value to fill the array with, default is 0.

Returns
None

add_scalar_property(*name: str*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Add a scalar property to the class.

This method initializes a new scalar property for the class instance. The property is stored as a 1-D NumPy array (scalar / entry) with a specified data type and default value.

Parameters

- **name** (*str*) – The name of the scalar property to be added.

- **dtype** (*data-type, optional*) – The desired data type for the property. Default is np.uint32.
- **default** (*scalar, optional*) – The default value for the property. Default is 0.

Returns
None

`add_vector_property`(*name: str*, *length: int*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Adds a vector property to the object.

This method initializes a new property with the given name as a 2-D NumPy array (vector per entry).

The array will have a shape of (length, self._capacity) and will be filled with the specified default value. The data type of the array elements is determined by the *dtype* parameter.

Parameters

- **name** (*str*) – The name of the property to be added.
- **length** (*int*) – The length of the vector.
- **dtype** (*data-type, optional*) – The desired data-type for the array, default is np.uint32.
- **default** (*scalar, optional*) – The default value to fill the array with, default is 0.

Returns
None

`property capacity: int`

Returns the capacity of the laser frame (total possible entries for dynamic properties).

Returns
int – The capacity of the laser frame.

`property count: int`

Returns the current count (equivalent to len()).

Returns
int – The current count value.

`describe`(*target=None*) → str

Return a formatted string description of the laserframe object, including its attributes and their values.

Parameters
target (*string*) – Optional string for the report header (generally the name of the LaserFrame variable, e.g., "People". Unlike functions, we can't get the name of a variable automatically).

Returns

str –

A formatted string describing the laserframe object, including its capacity, count, and details of its scalar, vector, and other properties.

`classmethod load_snapshot`(*path*, *n_ppl*, *cbr*, *nt*)

Load a LaserFrame and optional extras from an HDF5 snapshot file.

Parameters

- **path** (*str*) – Path to the HDF5 snapshot file.
- **n_ppl** (*float or array-like*) – Original total population (or per-node array) used to estimate births.
- **cbr** (*float or array-like*) – Crude birth rate (per 1000/year).

- **nt** (*int*) – Simulation duration (number of ticks).

Returns

frame (LaserFrame) – Loaded LaserFrame object. results_r (np.ndarray or None): Optional 2D numpy array of recovered counts. pars (dict or None): Optional dictionary of parameters.

`save_snapshot`(*path*, *results_r=None*, *pars=None*)

Save this LaserFrame and optional extras to an HDF5 snapshot file.

Parameters

- **path** (*Path*) – Destination file path
- **results_r** (*np.ndarray*) – Optional 2D numpy array of recovered counts
- **pars** (*PropertySet or dict*) – Optional PropertySet or dict of parameters

`sort`(*indices*, *verbose: bool = False*) → None

Sorts the elements of the object's numpy arrays based on the provided indices.

Parameters

- **indices** (*np.ndarray*) – An array of indices used to sort the numpy arrays. Must be of integer type and have the same length as the frame count (*self._count*).
- **verbose** (*bool, optional*) – If True, prints the sorting progress for each numpy array attribute. Defaults to False.

Raises

`AssertionError` – If *indices* is not an integer array or if its length does not match the frame count of active elements.

`squash`(*indices*, *verbose: bool = False*) → None

Reduces the active count of the internal numpy arrays keeping only elements True in the provided boolean indices.

Parameters

- **indices** (*np.ndarray*) – A boolean array indicating which elements to keep. Must have the same length as the current frame active element count.
- **verbose** (*bool, optional*) – If True, prints detailed information about the squashing process. Defaults to False.

Raises

`AssertionError` – If *indices* is not a boolean array or if its length does not match the current frame active element count.

Returns

None

5.1.7 laser.core.migration module

This module provides various functions to calculate migration networks based on different models, including the gravity model, competing destinations model, Stouffer's model, and the radiation model.

Additionally, it includes a utility function to calculate the great-circle distance between two points on the Earth's surface using the Haversine formula.

Functions:

gravity(pops: np.ndarray, distances: np.ndarray, k: float, a: float, b: float, c: float, max_frac: Union[float, None]=None, kwargs) -> np.ndarray:
Compute a gravity-model migration network based on origin and destination populations and pairwise distances.

row_normalizer(network: np.ndarray, max_rowsum: float) -> np.ndarray:
Normalize the rows of a given network matrix such that no row sum exceeds a specified maximum value.

competing_destinations(pops: np.ndarray, distances: np.ndarray, b: float, c: float, delta: float, params) -> np.ndarray:
Compute a migration network using the competing-destinations model, incorporating the influence of alternative destination nodes.

stouffer(pops: np.ndarray, distances: np.ndarray, k: float, a: float, b: float, include_home: bool, params) -> np.ndarray:
Compute a migration network using a modified Stouffer's model.

radiation(pops: np.ndarray, distances: np.ndarray, k: float, include_home: bool, params) -> np.ndarray:
Compute a migration network using the radiation model, which models flows based on intervening population rather than physical distance.

distance(lat1: float, lon1: float, lat2: float, lon2: float) -> float:
Calculate the great-circle distance between two points on the Earth's surface using the Haversine formula.

`laser.core.migration.`**`competing_destinations`**(*pops*, *distances*, *k*, *a*, *b*, *c*, *delta*, ***params*)

Calculate the competing destinations model for a given set of populations and distances. (Fotheringham AS. Spatial flows and spatial patterns. Environment and planning A. 1984;16(4):529-543)

This function computes a network matrix based on the gravity model and then adjusts it using the competing destinations model. The adjustment is done by considering the interference from other destinations.

Mathematical formula:

Element-by-element:
$$ network_{i,j} = k times p_i^a times p_j^b / distance_{i,j}^c times sum_k {(p_k^b / distance_{j,k}^c text {small for k not in [i,j]})^{delta} } $$

As-implemented numpy math:

- Compute all terms up to the sum_k using the gravity model
- Construct the matrix inside the sum: `p**b * distances**(1-c)`
- **Sum on the second axis (k), and subtract off the diagonal (j=k terms):**
  ```
  row_sums = np.sum(competition_matrix, axis=1) - np.
  diag(competition_matrix)
  ```
- **Now element-by-element, subtract k=i terms off the sum, exponentiate, and multiply the original network term:**
  ```
  network[i][j] = network[i][j] * (row_sums[i] - competition_matrix[i][j])
  ** delta
  ```

Parameters

- **pops** (*numpy.ndarray*) – Array of populations.
- **distances** (*numpy.ndarray*) – Array of distances between locations.
- **k** (*float*) – Scaling constant.
- **a** (*float*) – Exponent for the population of the origin.
- **b** (*float*) – Exponent parameter for populations in the gravity model.

- **c** (*float*) – Exponent parameter for distances in the gravity model.
- **delta** (*float*) – Exponent parameter for the competing destinations adjustment.
- **params** (*dict*) – Additional parameters to be passed to the gravity model.

Returns
network (numpy.ndarray) – Adjusted network matrix based on the competing destinations model.

`laser.core.migration.`**`distance`**(*lat1*, *lon1*, *lat2=None*, *lon2=None*)

Calculate the great-circle distance between two points on the Earth's surface. This function uses the Haversine formula to compute the distance between two points specified by their latitude and longitude in decimal degrees.

- If *lat2* and *lon2* are not provided, they default to *lat1* and *lon1*, respectively. This supports the default case of calculating the NxN matrix of distances between all pairs of points in (*lat1*, ` lon1`).
- If all arguments are scalars, will return a single scalar distance, (*lat1*, *lon1*) to (*lat2*, *lon2*).
- If *lat2*, *lon2* are vectors, will return a vector of distances, (*lat1*, *lon1*) to each lat/lon in *lat2*, *lon2*.
- If *lat1*, *lon1* and *lat2*, *lon2* are vectors, will return a matrix with shape (N, M) of distances where N is the length of *lat1*/*lon1* and M is the length of *lat2*/*lon2*.

Parameters

- **lat1** (*float*) – Latitude of the first point(s) in decimal degrees [-90, 90].
- **lon1** (*float*) – Longitude of the first point(s) in decimal degrees [-180, 180].
- **lat2** (*float*) – Latitude of the second point(s) in decimal degrees [-90, 90].
- **lon2** (*float*) – Longitude of the second point(s) in decimal degrees [-180, 180].

Returns
distance (float) – The distance between the points in kilometers.

`laser.core.migration.`**`gravity`**(*pops: ndarray*, *distances: ndarray*, *k: float*, *a: float*, *b: float*, *c: float*, ***kwargs*)

Calculate a gravity model network.

This function computes a gravity model network based on the provided populations and distances. The gravity model estimates migration or interaction flows between populations using a mathematical formula that incorporates scaling, population sizes, and distances.

Mathematical formula::
$$ network_{i,j} = k \cdot \frac{p_i^a \cdot p_j^b}{distance_{i,j}^c} $$

As implemented in NumPy:

```
network = k * (pops[:, np.newaxis] ** a) * (pops ** b) * (distances ** (-1 * c))
```

Parameters

- **pops** (*numpy.ndarray*) – 1D array of population sizes for each node.
- **distances** (*numpy.ndarray*) – 2D array of distances between nodes. Must be symmetric, with self-distances (diagonal) handled.
- **k** (*float*) – Scaling constant to adjust the overall magnitude of interaction flows.
- **a** (*float*) – Exponent for the population size of the origin node.
- **b** (*float*) – Exponent for the population size of the destination node.

- **c** (*float*) – Exponent for the distance between nodes, controlling how distance impacts flows.
- **kwargs** (*dict*) – Additional keyword arguments (not used in the current implementation).

Returns

network (numpy.ndarray) – A 2D matrix representing the interaction network, where each element *network[i, j]* corresponds to the flow from node *i* to node *j*.

Example usage:

```
import numpy as np
from gravity_model import gravity

populations = np.array([1000, 500, 200])
distances = np.array([
    [0, 2, 3],
    [2, 0, 1],
    [3, 1, 0]
])

k = 0.5
a = 1.0
b = 1.0
c = 2.0

migration_network = gravity(populations, distances, k, a, b, c)

print("Migration Network:")
print(migration_network)
```

Notes

- The diagonal of the distances matrix is set to 1 internally to avoid division by zero.
- The diagonal of the output network is set to 0 to represent no self-loops.
- Ensure the distances matrix is symmetric and non-negative.

`laser.core.migration.`**`radiation`**(*pops*, *distances*, *k*, *include_home*, ***params*)

Calculate the migration network using the radiation model.

(Simini F, González MC, Maritan A, Barabási AL. A universal model for mobility and migration patterns. Nature. 2012;484(7392):96-100.)

Mathematical formula:

Element-by-element:

$$ network_{i,j} = k times p_i^a times (p_j / sum_k {p_k})^b, $$ where the sum proceeds over all k such that $distances_{i,k} leq distances_{i,j}$ the parameter `include_home` determines whether p_i is included or excluded from the sum.

As-implemented numpy math:

- Sort each row of the distance matrix (we'll use ' below to indicate distance-sorted vectors)
- Loop over "source nodes" i
  - Cumulative sum the sorted populations, ensuring appropriate handling when there are multiple destinations equidistant from the source

- Subtract the source node population if `include_home` is `False`
- **Construct the row of the network matrix as**
  $$ k times p_i times p_{j'} / (p_i + sum_{k'} {p_{k'}}) / (p_i + p_{j'} + sum_{k'} {p_{k'}}) $$

- Unsort the rows of the network

Parameters

- **pops** (*numpy.ndarray*) – Array of population sizes for each node.
- **distances** (*numpy.ndarray*) – 2D array of distances between nodes.
- **k** (*float*) – Scaling factor for the migration rates.
- **include_home** (*bool*) – Whether to include the home population in the calculations.
- **params** (*dict*) – Additional parameters (currently not used).

Returns

network (numpy.ndarray) – 2D array representing the migration network.

`laser.core.migration.`**`row_normalizer`**(*network*, *max_rowsum*)

Normalizes the rows of a given network matrix such that no row sum exceeds a specified maximum value.

Parameters

- **network** (*numpy.ndarray*) – A 2D array representing the network matrix.
- **max_rowsum** (*float*) – The maximum allowable sum for any row in the network matrix.

Returns

network (numpy.ndarray) – The normalized network matrix where no row sum exceeds the specified maximum value.

`laser.core.migration.`**`stouffer`**(*pops*, *distances*, *k*, *a*, *b*, *include_home*, ***params*)

Computes a migration network using a modified Stouffer's model.

(Stouffer SA. Intervening opportunities: a theory relating mobility and distance. American sociological review. 1940;5(6):845-867)

Mathematical formula:

Element-by-element:
$$ network_{i,j} = k times p_i times p_j / ((p_i + sum_k {p_k}) (p_i + p_j + sum_k {p_k})) $$ the parameter `include_home` determines whether p_i is included or excluded from the sum

As-implemented numpy math:

- Sort each row of the distance matrix (we'll use ' below to indicate distance-sorted vectors)
- Loop over "source nodes" i:
  - Cumulative sum the sorted populations, ensuring appropriate handling when there are multiple destinations equidistant from the source
  - Subtract the source node population if `include_home` is `False`
  - Construct the row of the network matrix as $k times p_i^a times (p_{j'} / sum_{k'} {p_{k'}})^b$
- Unsort the rows of the network

Parameters

- **pops** (*numpy.ndarray*) – An array of population sizes.
- **distances** (*numpy.ndarray*) – A 2D array where *distances[i][j]* is the distance from location i to location j.
- **k** (*float*) – A scaling factor for the migration rates.
- **a** (*float*) – Exponent applied to the population size of the origin.
- **b** (*float*) – Exponent applied to the ratio of destination population to the sum of all populations at equal or lesser distances.
- **include_home** (*bool*) – If True, includes the home population in the cumulative sum; otherwise, excludes it.
- ****params** – Additional parameters (not used in the current implementation).

Returns

network (numpy.ndarray) – A 2D array representing the migration network, where *network[i][j]* is the migration rate from location i to location j.

`laser.core.migration.`**`sum_populations_as_close_or_closer`**`(`*sorted_pops*, *sorted_distance_row*`)`

5.1.8 laser.core.propertyset module

Implements a PropertySet class that can be used to store properties in a dictionary-like object.

`class laser.core.propertyset.`**`PropertySet`**`(`**bags:* PropertySet | *list* | *tuple* | *dict*`)`

Bases: `object`

A class that can be used to store properties in a dictionary-like object with *.property* access to properties.

Examples

Basic Initialization:

from laser.core import PropertySet ps = PropertySet() ps[‘infection_status’] = ‘infected’ ps[‘age’] = 35 print(ps.infection_status) # Outputs: ‘infected’ print(ps[‘age’]) # Outputs: 35

Combining two PropertySets:

ps1 = PropertySet({‘immunity’: ‘high’, ‘region’: ‘north’}) ps2 = PropertySet({‘infectivity’: 0.7}) combined_ps = ps1 + ps2 print(combined_ps.to_dict()) # Outputs: {‘immunity’: ‘high’, ‘region’: ‘north’, ‘infectivity’: 0.7}

Creating a PropertySet from a dictionary:

ps = PropertySet({‘mything’: 0.4, ‘that_other_thing’: 42}) print(ps.mything) # Outputs: 0.4 print(ps.that_other_thing) # Outputs: 42 print(ps.to_dict()) # Outputs: {‘mything’: 0.4, ‘that_other_thing’: 42}

Save and load:

ps.save(‘properties.json’) loaded_ps = PropertySet.load(‘properties.json’) print(loaded_ps.to_dict()) # Outputs the saved properties

Property access and length:

ps[‘status’] = ‘susceptible’ ps[‘exposure_timer’] = 5 print(ps[‘status’]) # Outputs: ‘susceptible’ print(len(ps)) # Outputs: 4

In-Place addition (added keys must *not* exist in the destination PropertySet):

ps += {'new_timer': 10, 'susceptibility': 0.75} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 5, 'new_timer': 10, 'susceptibility': 0.75}

In-place update (keys *must* already exist in the destination PropertySet):

ps <<= {'exposure_timer': 10, 'infectivity': 0.8} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 10, 'infectivity': 0.8}

In-place addition or update (no restriction on incoming keys):

ps |= {'new_timer': 10, 'exposure_timer': 8} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 8, 'new_timer': 10}

`__add__`(*other*)

Add another PropertySet to this PropertySet.

This method allows the use of the + operator to combine two PropertySet instances.

Parameters
other (*PropertySet*) – The other PropertySet instance to add.

Returns
PropertySet (PropertySet) – A new PropertySet instance that combines the properties of both instances.

`__contains__`(*key*)

Check if a key is in the property set.

Parameters
key (*str*) – The key to check for existence in the property set.

Returns
bool (bool) – True if the key exists in the property set, False otherwise.

`__eq__`(*other*)

Check if two PropertySet instances are equal.

Parameters
other (*PropertySet*) – The other PropertySet instance to compare.

Returns
bool (bool) – True if the two instances are equal, False otherwise.

`__getitem__`(*key*)

Retrieve the attribute of the object with the given key (e.g., `ps[key]`).

Parameters
key (*str*) – The name of the attribute to retrieve.

Returns
Any (any) – The value of the attribute with the specified key.

Raises
`AttributeError` – If the attribute with the specified key does not exist.

`__iadd__`(*other*)

Implements the in-place addition (+=) operator for the class.

This method allows the instance to be updated with attributes from another instance of the same class or from a dictionary. If *other* is an instance of the same class, its attributes are copied to the current instance. If *other* is a dictionary, its key-value pairs are added as attributes to the current instance.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary to add to the current instance.

Returns
self (PropertySet) – The updated instance with the new attributes.

Raises

- `AssertionError` – If *other* is neither an instance of the same class nor a dictionary.
- `ValueError` – If *other* contains keys already present in the PropertySet.

`__ilshift__`(*other*)

Implements the <<= operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
self (PropertySet) – The updated instance with the overrides from other.

Raises

- `AssertionError` – If *other* is neither an instance of the same class nor a dictionary.
- `ValueError` – If *other* contains keys not present in the PropertySet.

`__ior__`(*other*)

Implements the |= operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
self (PropertySet) – The updated instance with all the values of self with new or overriding values from other.

Raises
`AssertionError` – If *other* is neither an instance of the same class nor a dictionary.

`__len__`()

Return the number of attributes in the instance.

This method returns the number of attributes stored in the instance's __dict__ attribute, which represents the instance's namespace.

Returns
int (int) – The number of attributes in the instance.

`__lshift__`(*other*)

Implements the << operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
PropertySet (PropertySet) – A new PropertySet with all the values of the first PropertySet with overrides from the second PropertySet.

Raises

- `AssertionError` – If *other* is neither an instance of the same class nor a dictionary.
- `ValueError` – If *other* contains keys not present in the PropertySet.

`__or__`(*other*)

Implements the | operator on PropertySet to add new or override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
PropertySet (PropertySet) – A new PropertySet with all the values of the first PropertySet with new or overriding values from the second PropertySet.

Raises
`AssertionError` – If *other* is neither an instance of the same class nor a dictionary.

`__repr__`() → str

Return a string representation of the PropertySet instance.

The string representation includes the class name and the dictionary of the instance's attributes.

Returns
str – A string representation of the PropertySet instance.

`__setitem__`(*key*, *value*)

Set the value of an attribute. This method allows setting an attribute of the instance using the dictionary-like syntax (e.g., `ps[key] = value`).

Parameters

- **key** (*str*) – The name of the attribute to set.
- **value** (*any*) – The value to set for the attribute.

`__str__`() → str

Returns a string representation of the object's dictionary.

This method is used to provide a human-readable string representation of the object, which includes all the attributes stored in the object's *__dict__*.

Returns
str – A string representation of the object's dictionary.

`static load`(*filename*)

Load a PropertySet from a specified file.

Parameters
filename (*str*) – The path to the file where the PropertySet is saved.

Returns
PropertySet (PropertySet) – The PropertySet instance loaded from the file.

`save`(*filename*)

Save the PropertySet to a specified file.

Parameters
filename (*str*) – The path to the file where the PropertySet will be saved.

`to_dict`()

Convert the PropertySet to a dictionary.

5.1.9 laser.core.random module

Functions for seeding and accessing the laser-core random number generator.

Using the seed() function here and the pseudo-random number generator (PRNG) returned from the prng() function in simulation code will guarantee that the same random number stream is generated and used during simulation runs using the same seed value (assuming no changes to code which add or remove PRNG calls or change the number of random draws requested). This is important for reproducibility and debugging purposes.

`laser.core.random.get_seed()` → uint32

Return the seed used to initialize the pseudo-random number generator.

Returns
uint32 – The seed value used to initialize the random number generators.

`laser.core.random.prng()` → Generator

Return the global (to LASER) pseudo-random number generator.

`laser.core.random.seed(seed)` → Generator

Initialize the pseudo-random number generator with a given seed.

This function sets the global pseudo-random number generator (_prng) to a new instance of numpy's default random generator initialized with the provided seed. It also seeds Numba's per-thread random number generators with the same seed.

Parameters
seed (*int*) – The seed value to initialize the random number generators.

Returns
numpy.random.Generator – The initialized pseudo-random number generator.

5.1.10 laser.core.utils module

This module provides utility functions for the laser-measles project.

Functions:

calc_capacity(birthrates: np.ndarray, initial_pop: np.ndarray, safety_factor: float = 1.0) -> np.ndarray:
Calculate the population capacity after a given number of ticks based on a constant birth rate.

grid(shape: Tuple[int, int], fill_value: float = 0.0) -> np.ndarray:
Create a 2D grid (numpy array) of the specified shape, filled with the given value.

`laser.core.utils.calc_capacity(birthrates: ndarray, initial_pop: ndarray, safety_factor: float = 1.0)` → ndarray

Estimate the required capacity (number of agents) to model a population given birthrates over time.

Parameters

- **birthrates** (*np.ndarray*) – 2D array of shape (nsteps, nnodes) representing birthrates (CBR) per 1,000 individuals per year.
- **initial_pop** (*np.ndarray*) – 1D array of length nnodes representing the initial population at each node.
- **safety_factor** (*float*) – Safety factor to account for variability in population growth. Default is 1.0.

Returns
np.ndarray – 1D array of length nnodes representing the estimated required capacity (number of agents) at each node.

laser.core.utils.**grid**(*M=5*, *N=5*, *node_size_degs=0.08983*, *population_fn=None*, *origin_x=0*, *origin_y=0*, *states=None*)

Create an MxN grid of cells anchored at (lat, long) with populations and geometries.

By default all nodes are initialized with the full population in the first state (e.g., "S" for susceptible).

Parameters

- **M** (*int*) – Number of rows (north-south).
- **N** (*int*) – Number of columns (east-west).
- **node_size_degs** (*float*) – Size of each cell in decimal degrees (default 0.08983 10km at the equator).
- **population_fn** (*callable*) – Function(row, col) returning population for a cell. Default is uniform random between 1,000 and 100,000.
- **origin_x** (*float*) – longitude of the origin in decimal degrees (bottom-left corner) -180 <= origin_x < 180.
- **origin_y** (*float*) – latitude of the origin in decimal degrees (bottom-left corner) -90 <= origin_y < 90.
- **states** (*list*) – List of state names to initialize in the GeoDataFrame. Default is ["S", "E", "I", "R"].

Returns

scenario (GeoDataFrame) – Columns are nodeid, population, geometry.

laser.core.utils.**initialize_population**(*grid*, *initial: list | ndarray*, *states=None*)

Initialize the population states in the grid based on the initial state counts provided.

Provide integer values to set the exact counts for each state at each node. Alternatively, provide fractional values between 0.0 and 1.0 to set proportions of the population for each state at each node. In the latter case, the first state in the states list will be computed as the remainder of the population after assigning the other states.

Parameters

- **grid** (*GeoDataFrame*) – The grid GeoDataFrame with population and state columns.
- **initial** (*list or np.ndarray*) – A list or array of shape (1|nnodes, nstates) representing the initial counts for each state at each node. If the shape is (1, nstates), the same initial state distribution will be applied to all nodes.
- **states** (*list*) – List of state names corresponding to the columns in the grid. Default is ["S", "E", "I", "R"].

Returns

GeoDataFrame – The updated grid with initialized population states.

5.1.11 Module contents

class laser.core.**LaserFrame**(*capacity: int*, *initial_count: int = -1*, ***kwargs*)

Bases: `object`

The LaserFrame class, similar to a db table or a Pandas DataFrame, holds dynamically allocated data for agents (generally 1-D or scalar) or for nodes|patches (e.g., 1-D for scalar value per patch or 2-D for time-varying per patch).

add(*count: int*) → tuple[int, int]

Adds the specified count to the current count of the LaserFrame.

This method increments the internal count by the given count, ensuring that the total does not exceed the frame's capacity. If the addition would exceed the capacity, an assertion error is raised. This method is typically used to add new births during the simulation.

Parameters
count (*int*) – The number to add to the current count.

Returns
tuple[int, int] – A tuple containing the [start index, end index) after the addition.

Raises
AssertionError – If the resulting count exceeds the frame's capacity.

add_array_property(*name: str*, *shape: tuple*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Adds an array property to the object.

This method initializes a new property with the given name as a multi-dimensional NumPy array.

The array will have the given shape (note that there is no implied dimension of size self._capacity), datatype (default is np.uint32), and default value (default is 0).

Parameters

- **name** (*str*) – The name of the property to be added.
- **shape** (*tuple*) – The shape of the array.
- **dtype** (*data-type, optional*) – The desired data-type for the array, default is np.uint32.
- **default** (*scalar, optional*) – The default value to fill the array with, default is 0.

Returns
None

add_scalar_property(*name: str*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Add a scalar property to the class.

This method initializes a new scalar property for the class instance. The property is stored as a 1-D NumPy array (scalar / entry) with a specified data type and default value.

Parameters

- **name** (*str*) – The name of the scalar property to be added.
- **dtype** (*data-type, optional*) – The desired data type for the property. Default is np.uint32.
- **default** (*scalar, optional*) – The default value for the property. Default is 0.

Returns
None

add_vector_property(*name: str*, *length: int*, *dtype=<class 'numpy.uint32'>*, *default=0*) → None

Adds a vector property to the object.

This method initializes a new property with the given name as a 2-D NumPy array (vector per entry).

The array will have a shape of (length, self._capacity) and will be filled with the specified default value. The data type of the array elements is determined by the *dtype* parameter.

Parameters

- **name** (*str*) – The name of the property to be added.

- **length** (*int*) – The length of the vector.
- **dtype** (*data-type, optional*) – The desired data-type for the array, default is np.uint32.
- **default** (*scalar, optional*) – The default value to fill the array with, default is 0.

Returns
None

`property capacity: int`

Returns the capacity of the laser frame (total possible entries for dynamic properties).

Returns
int – The capacity of the laser frame.

`property count: int`

Returns the current count (equivalent to len()).

Returns
int – The current count value.

`describe`(*target=None*) → str

Return a formatted string description of the laserframe object, including its attributes and their values.

Parameters
target (*string*) – Optional string for the report header (generally the name of the LaserFrame variable, e.g., "People". Unlike functions, we can't get the name of a variable automatically).

Returns

str –

A formatted string describing the laserframe object, including its capacity, count, and details of its scalar, vector, and other properties.

`classmethod load_snapshot`(*path*, *n_ppl*, *cbr*, *nt*)

Load a LaserFrame and optional extras from an HDF5 snapshot file.

Parameters

- **path** (*str*) – Path to the HDF5 snapshot file.
- **n_ppl** (*float or array-like*) – Original total population (or per-node array) used to estimate births.
- **cbr** (*float or array-like*) – Crude birth rate (per 1000/year).
- **nt** (*int*) – Simulation duration (number of ticks).

Returns
frame (LaserFrame) – Loaded LaserFrame object. results_r (np.ndarray or None): Optional 2D numpy array of recovered counts. pars (dict or None): Optional dictionary of parameters.

`save_snapshot`(*path*, *results_r=None*, *pars=None*)

Save this LaserFrame and optional extras to an HDF5 snapshot file.

Parameters

- **path** (*Path*) – Destination file path
- **results_r** (*np.ndarray*) – Optional 2D numpy array of recovered counts
- **pars** (*PropertySet or dict*) – Optional PropertySet or dict of parameters

`sort`(*indices*, *verbose: bool = False*) → None

Sorts the elements of the object's numpy arrays based on the provided indices.

Parameters

- **indices** (*np.ndarray*) – An array of indices used to sort the numpy arrays. Must be of integer type and have the same length as the frame count (*self._count*).
- **verbose** (*bool, optional*) – If True, prints the sorting progress for each numpy array attribute. Defaults to False.

Raises

`AssertionError` – If *indices* is not an integer array or if its length does not match the frame count of active elements.

`squash`(*indices*, *verbose: bool = False*) → None

Reduces the active count of the internal numpy arrays keeping only elements True in the provided boolean indices.

Parameters

- **indices** (*np.ndarray*) – A boolean array indicating which elements to keep. Must have the same length as the current frame active element count.
- **verbose** (*bool, optional*) – If True, prints detailed information about the squashing process. Defaults to False.

Raises

`AssertionError` – If *indices* is not a boolean array or if its length does not match the current frame active element count.

Returns

None

`class laser.core.PropertySet`(**bags:* PropertySet | *list* | *tuple* | *dict*)

Bases: `object`

A class that can be used to store properties in a dictionary-like object with *.property* access to properties.

Examples

Basic Initialization:

from laser.core import PropertySet ps = PropertySet() ps['infection_status'] = 'infected' ps['age'] = 35 print(ps.infection_status) # Outputs: 'infected' print(ps['age']) # Outputs: 35

Combining two PropertySets:

ps1 = PropertySet({'immunity': 'high', 'region': 'north'}) ps2 = PropertySet({'infectivity': 0.7}) combined_ps = ps1 + ps2 print(combined_ps.to_dict()) # Outputs: {'immunity': 'high', 'region': 'north', 'infectivity': 0.7}

Creating a PropertySet from a dictionary:

ps = PropertySet({'mything': 0.4, 'that_other_thing': 42}) print(ps.mything) # Outputs: 0.4 print(ps.that_other_thing) # Outputs: 42 print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42}

Save and load:

ps.save('properties.json') loaded_ps = PropertySet.load('properties.json') print(loaded_ps.to_dict()) # Outputs the saved properties

Property access and length:

ps['status'] = 'susceptible' ps['exposure_timer'] = 5 print(ps['status']) # Outputs: 'susceptible' print(len(ps)) # Outputs: 4

In-Place addition (added keys must *not* exist in the destination PropertySet):

ps += {'new_timer': 10, 'susceptibility': 0.75} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 5, 'new_timer': 10, 'susceptibility': 0.75}

In-place update (keys *must* already exist in the destination PropertySet):

ps <<= {'exposure_timer': 10, 'infectivity': 0.8} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 10, 'infectivity': 0.8}

In-place addition or update (no restriction on incoming keys):

ps |= {'new_timer': 10, 'exposure_timer': 8} print(ps.to_dict()) # Outputs: {'mything': 0.4, 'that_other_thing': 42, 'status': 'susceptible', 'exposure_timer': 8, 'new_timer': 10}

`__add__`(*other*)

Add another PropertySet to this PropertySet.

This method allows the use of the + operator to combine two PropertySet instances.

Parameters
other (*PropertySet*) – The other PropertySet instance to add.

Returns
PropertySet (PropertySet) – A new PropertySet instance that combines the properties of both instances.

`__contains__`(*key*)

Check if a key is in the property set.

Parameters
key (*str*) – The key to check for existence in the property set.

Returns
bool (bool) – True if the key exists in the property set, False otherwise.

`__eq__`(*other*)

Check if two PropertySet instances are equal.

Parameters
other (*PropertySet*) – The other PropertySet instance to compare.

Returns
bool (bool) – True if the two instances are equal, False otherwise.

`__getitem__`(*key*)

Retrieve the attribute of the object with the given key (e.g., `ps[key]`).

Parameters
key (*str*) – The name of the attribute to retrieve.

Returns
Any (any) – The value of the attribute with the specified key.

Raises
`AttributeError` – If the attribute with the specified key does not exist.

__iadd__(*other*)

Implements the in-place addition (+=) operator for the class.

This method allows the instance to be updated with attributes from another instance of the same class or from a dictionary. If *other* is an instance of the same class, its attributes are copied to the current instance. If *other* is a dictionary, its key-value pairs are added as attributes to the current instance.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary to add to the current instance.

Returns
self (PropertySet) – The updated instance with the new attributes.

Raises

- **AssertionError** – If *other* is neither an instance of the same class nor a dictionary.
- **ValueError** – If *other* contains keys already present in the PropertySet.

__ilshift__(*other*)

Implements the <<= operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
self (PropertySet) – The updated instance with the overrides from other.

Raises

- **AssertionError** – If *other* is neither an instance of the same class nor a dictionary.
- **ValueError** – If *other* contains keys not present in the PropertySet.

__ior__(*other*)

Implements the |= operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
self (PropertySet) – The updated instance with all the values of self with new or overriding values from other.

Raises
AssertionError – If *other* is neither an instance of the same class nor a dictionary.

__len__()

Return the number of attributes in the instance.

This method returns the number of attributes stored in the instance's __dict__ attribute, which represents the instance's namespace.

Returns
int (int) – The number of attributes in the instance.

__lshift__(*other*)

Implements the << operator on PropertySet to override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
PropertySet (PropertySet) – A new PropertySet with all the values of the first PropertySet with overrides from the second PropertySet.

Raises
- **AssertionError** – If *other* is neither an instance of the same class nor a dictionary.
- **ValueError** – If *other* contains keys not present in the PropertySet.

__or__(*other*)

Implements the | operator on PropertySet to add new or override existing values with new values.

Parameters
other (*Union[type(self), dict]*) – The object or dictionary with overriding values.

Returns
PropertySet (PropertySet) – A new PropertySet with all the values of the first PropertySet with new or overriding values from the second PropertySet.

Raises
AssertionError – If *other* is neither an instance of the same class nor a dictionary.

__repr__() → str

Return a string representation of the PropertySet instance.

The string representation includes the class name and the dictionary of the instance's attributes.

Returns
str – A string representation of the PropertySet instance.

__setitem__(*key*, *value*)

Set the value of an attribute. This method allows setting an attribute of the instance using the dictionary-like syntax (e.g., `ps[key] = value`).

Parameters
- **key** (*str*) – The name of the attribute to set.
- **value** (*any*) – The value to set for the attribute.

__str__() → str

Returns a string representation of the object's dictionary.

This method is used to provide a human-readable string representation of the object, which includes all the attributes stored in the object's *__dict__*.

Returns
str – A string representation of the object's dictionary.

static load(*filename*)

Load a PropertySet from a specified file.

Parameters
filename (*str*) – The path to the file where the PropertySet is saved.

Returns
PropertySet (PropertySet) – The PropertySet instance loaded from the file.

save(*filename*)

Save the PropertySet to a specified file.

Parameters
filename (*str*) – The path to the file where the PropertySet will be saved.

`to_dict()`

Convert the PropertySet to a dictionary.

`class laser.core.SortedQueue(capacity: int, values: ndarray)`

Bases: `object`

A sorted (priority) queue implemented using NumPy arrays and sped-up with Numba.

Using the algorithm from the Python heapq module.

__init__ with an existing array of sorting values

__push__ with an index into sorting values

__pop__ returns the index of the lowest sorting value and its value

`__len__()` → int

Return the number of elements in the sorted queue.

Returns
int – The number of elements in the sorted queue.

`peeki()` → uint32

Returns the index of the smallest value element in the sorted queue without removing it.

Raises
`IndexError` – If the sorted queue is empty.

Returns
np.uint32 – The index of the smallest value element.

`peekiv()` → tuple[uint32, Any]

Returns the index and value of the smallest value element in the sorted queue without removing it.

Returns
tuple[np.uint32, Any] – A tuple containing the index and value of the smallest value element.

Raises
`IndexError` – If the sorted queue is empty.

`peekv()` → Any

Return the smallest value from the sorted queue without removing it.

Raises
`IndexError` – If the sorted queue is empty.

Returns
Any – The value with the smallest value in the sorted queue.

`popi()` → uint32

Removes and returns the index of the smallest value element in the sorted queue.

This method first retrieves the index of the smallest value element using *peeki()*, then removes the element from the queue using *pop()*, and finally returns the index.

Returns
np.uint32 – The index of the smallest value element in the sorted queue.

`popiv()` → tuple[uint32, Any]

Removes and returns the index and value of the smallest value element in the sorted queue.

This method first retrieves the index and value of the smallest value element using *peekiv()*, then removes the element from the queue using *pop()*, and finally returns the index and value.

Returns
tuple[np.uint32, Any] – A tuple containing the index and value of the smallest value element.

popv() → Any

Removes and returns the value at the front of the sorted queue.

This method first retrieves the value at the front of the queue without removing it by calling *peekv()*, and then removes the front element from the queue by calling *pop()*. The retrieved value is then returned.

Returns
Any – The value at the front of the sorted queue.

push(*index*) → None

Insert an element into the sorted queue.

This method adds an element at the back of the sorted queue and then ensures the heap property is maintained by sifting the element forward to its correct position.

Parameters
index (*int*) – The index of the element to be added to the sorted queue.

Raises
IndexError – If the sorted queue is full.

laser.core.**compiled**(*args*)

CHAPTER

SIX

MIGRATION

Some things to note:

These models all take in some parameters, along with a vector of populations and distances between nodes, and spit out a matrix defining the connection between nodes.

- When comparing to other models, it's important to consider how implementation of the migration / connection model affects interpretation of the parameters. For example, a migration matrix could be implemented as a per-capita rate of travel from i to j, or as a total flux of people from i to j. If your migration model has a term that scales like p_i^a, using a per-capita rate introduces an implicit $+1$ into the exponent. Or, depending on how infectivity / mixing is handled locally, if the introduced infectivity is normalized to local population, that might introduce an ambiguity in interpreting the exponent on the destination population, is it b or $b - 1$, effectively? In the end, these are terms that could be calibrated away, but useful to keep this in mind for interpretation and comparison against other models - we aim to support users defining their own migration models and implementations, and this sort of ambiguity is important to keep in mind.
- I'm aiming to do most of this with element-by-element numpy functions when I can, though loops would probably translate more obviously to numba/c. I don't expect that computation of these matrices will be a substantial part of overall computational spend on a model either way.
- I have not tested nor written code to enforce conditions on the inputs. This should be done - e.g., populations coming in as integers can present wrapping issues when we start exponentiating and multiplying them (signed integers in particular can be a problem because you may wrap into negative numbers). So some input checking and such needs to be done.
- It's also worth investigating whether using large floats makes sense when computing these formulas, or whether we should put operations in a specific order. This is because depending on the choice of someone's metapop network and spatial model parameters, and the order of computations, we can end up multiplying, dividing, summing over numbers that can be across really different scales, so weirdness might happen with loss of precision? The integer issue above is more concerning and one that I have run into in the past.
- Distances on the diagonal of the distance matrix should always be 0. We should check for 0s elsewhere and throw an error. It's also nice to be able to use numpy element-by-element math without constant div-by-zero errors for the diagonal elements, so maybe each function should start by adding epsilon to the diagonal of the distance matrix? We're going to zero out those terms in the network anyway…

6.1 Gravity model

Functional form: $M_{i,j} = k\frac{p_i^a p_j^b}{d_{i,j}^c}$

Special cases of the gravity model (as noted above, both population exponents are subject to ± 1 ambiguity depending on implementation of spatial connectivity and local mixing):

- Xia's model: $a = 0$

- mean-field model: $c = 0, a = 1, b = 1$
- spatial diffusion: $a = 0, b = 0$

6.1.1 Example usage

Below is an example of how to use the gravity model to compute migration flows between populations located at specific distances. The example assumes unequal population sizes and calculates the number of migrants moving between nodes based on the gravity model.

```
import numpy as np
from laser.core.migration import gravity

# Define populations and distances
populations = np.array([5000, 10000, 15000, 20000, 25000])  # Unequal populations
distances = np.array([
    [0.0, 10.0, 15.0, 20.0, 25.0],
    [10.0, 0.0, 10.0, 15.0, 20.0],
    [15.0, 10.0, 0.0, 10.0, 15.0],
    [20.0, 15.0, 10.0, 0.0, 10.0],
    [25.0, 20.0, 15.0, 10.0, 0.0]
])

# Gravity model parameters
k = 0.1    # Scaling constant
a = 0.5    # Exponent for the population of the origin
b = 1.0    # Exponent for the population of the destination
c = 2.0    # Exponent for the distance

# Compute the gravity model network
migration_network = gravity(populations, distances, k=k, a=a, b=b, c=c)

# Normalize to ensure total migrations represent 1% of the population
total_population = np.sum(populations)
migration_fraction = 0.01  # 1% of the population migrates
scaling_factor = (total_population * migration_fraction) / np.sum(migration_network)
migration_network *= scaling_factor

# Generate a node ID array for agents
node_ids = np.concatenate([np.full(count, i) for i, count in enumerate(populations)])

# Initialize a 2D array for migration counts
migration_matrix = np.zeros_like(distances, dtype=int)

# Select migrants based on the gravity model
for origin in range(len(populations)):
    for destination in range(len(populations)):
        if origin != destination:
            # Number of migrants to move from origin to destination
            num_migrants = int(migration_network[origin, destination])
            # Select migrants randomly
            origin_ids = np.where(node_ids == origin)[0]
            selected_migrants = np.random.choice(origin_ids, size=num_migrants,␣
↪replace=False)
```

(continues on next page)

(continued from previous page)

```
            # Update the migration matrix
            migration_matrix[origin, destination] = num_migrants
```

This example demonstrates the end-to-end process of using the gravity model to calculate migration flows and randomly assign agents to those flows. The resulting migration matrix shows the number of individuals migrating between nodes.

6.2 Capping the total fraction of population that can migrate / infectivity that can be exported on a given timestep

Because the inputs to spatial models (populations, distances) can vary over many orders of magnitude, we can run into situations where a a small number of nodes, often those closest to but distinct from large population centers, will end up with huge outflows. The below illustrates an easy way to implement a standard gravity/radiation/etc. model, but cap the total amount of migration/infectivity outflow from any single metapopulation.

6.3 The Competing Destinations model

There are many models that aim to account for the impact of competition or synergy between potential destinations. Some aim to account for some "screening" effect of travel to distant destinations due to competition from attractive destinations closer to the origin i. This model, in contrast, (Fotheringham AS. Spatial flows and spatial patterns. Environment and Planning A. 1984;16(4):529–543) aims to account for effects from other attractive destinations near destination j; notably, this effect could be synergistic or antagonistic, depending on the sign of the exponent δ.

For example, in a "synergistic" version, perhaps migratory flow from Boston to Baltimore is higher than flow between two comparator cities of similar population and at similar distance, because the proximity of Washington, D.C. to Baltimore makes travel to Baltimore more attractive to Bostonians – this would be accounted for by a positive value of δ. On the other hand, this term may also be "antagonistic", if Washington is such an attractive destination that Bostonians eschew travel to Baltimore entirely; this would indicate a negative value of δ.

Mathematical Formulation: $M_{i,j} = k\frac{p_i^a p_j^b}{d_{i,j}^c}\left(\sum_{k\neq i,j}\frac{p_k^b}{d_{jk}^c}\right)^\delta$

6.4 Stouffer's rank model

Stouffer (Stouffer SA. Intervening opportunities: a theory relating mobility and distance. American Sociological Review. 1940;5(6):845–867) argued that human mobility patterns do not respond to absolute distance directly, but only indirectly through the accumulation of intervening opportunities for destinations. Stouffer thus proposed a model with no distance-dependence at all, rather only a term that accounts for all potential destinations closer than destination j; thus, longer-distance travel depends on the density of attractive destinations at shorter distances.

Mathematical formulation:

Define $\Omega(i,j)$ to be the set of all locations k such that $D_{i,k} \leq D_{i,j}$

$M_{i,j} = kp_i^a \sum_j \left(\frac{p_j}{\sum_{k\in\Omega(i,j)} p_k}\right)^b$

This presents us with the choice of whether or not the origin population i is included in Ω – i.e., does the same "gravity" that brings others to visit a community reduce the propensity of that community's members to travel to other communities?

The Stouffer model does not include impact from the local community: $\Omega(i,j) = (k : 0 < D_{i,k} \leq D_{i,j})$.

The Stouffer variant model does include the impact of the local community: $\Omega(i,j) = (k : 0 \leq D_{i,k} \leq D_{i,j})$.

Rather than implementing twice, this implementation of the Stouffer model will include a parameter "include_home."

6.5 Radiation model

The radiation model (Simini F, González MC, Maritan A, Barabási AL. A universal model for mobility and migration patterns. Nature. 2012;484(7392):96–100.) is a parameter-free model (up to an overall scaling constant for total migration flux), derived from arguments around job-related commuting but essentially capturing a situation in which outbound migration flux from origin to destination is enhanced by destination population and absorbed by the density of nearer destinations.

Mathematical formulation: With Ω defined as above in the Stouffer model,

$$M_{i,j} = k \frac{p_i p_j}{\left(p_i + \sum_{k \in \Omega(i,j)} p_k\right)\left(p_i + p_j + \sum_{k \in \Omega(i,j)} p_k\right)}$$

We again use the parameter "include_home" to determine whether or not location i is to be included in $\Omega(i, j)$.

CHAPTER

SEVEN

POPULATION PYRAMIDS

The *AliasedDistribution* class provides a way to sample from a set of options with unequal probabilities, e.g., a population pyramid.

The input to the *AliasedDistribution* constructor is an array of counts by bin as we would naturally get from a population pyramid (# of people in each age bin).

AliasedDistribution.sample() returns *bin indices* so it is up to the user to convert the values returned from *sample()* to actual ages.

Expected format of the population pyramid CSV file for *load_pyramid_csv()*:

```
Header: Age,M,F
start-end,#males,#females
start-end,#males,#females
start-end,#males,#females
...
start-end,#males,#females
max+,#males,#females
```

E.g.:

```
Age,M,F
0-4,9596708,9175309
5-9,10361680,9904126
10-14,10781688,10274310
15-19,11448281,10950664
...
90-94,757034,1281854
95-99,172530,361883
100+,27665,76635
```

load_pyramid_csv() returns a 4 column NumPy array with the following columns:

```
0 - Lower bound of age bin, inclusive
1 - Upper bound of age bin, inclusive
2 - number of males in the age bin
3 - number of females in the age bin
```

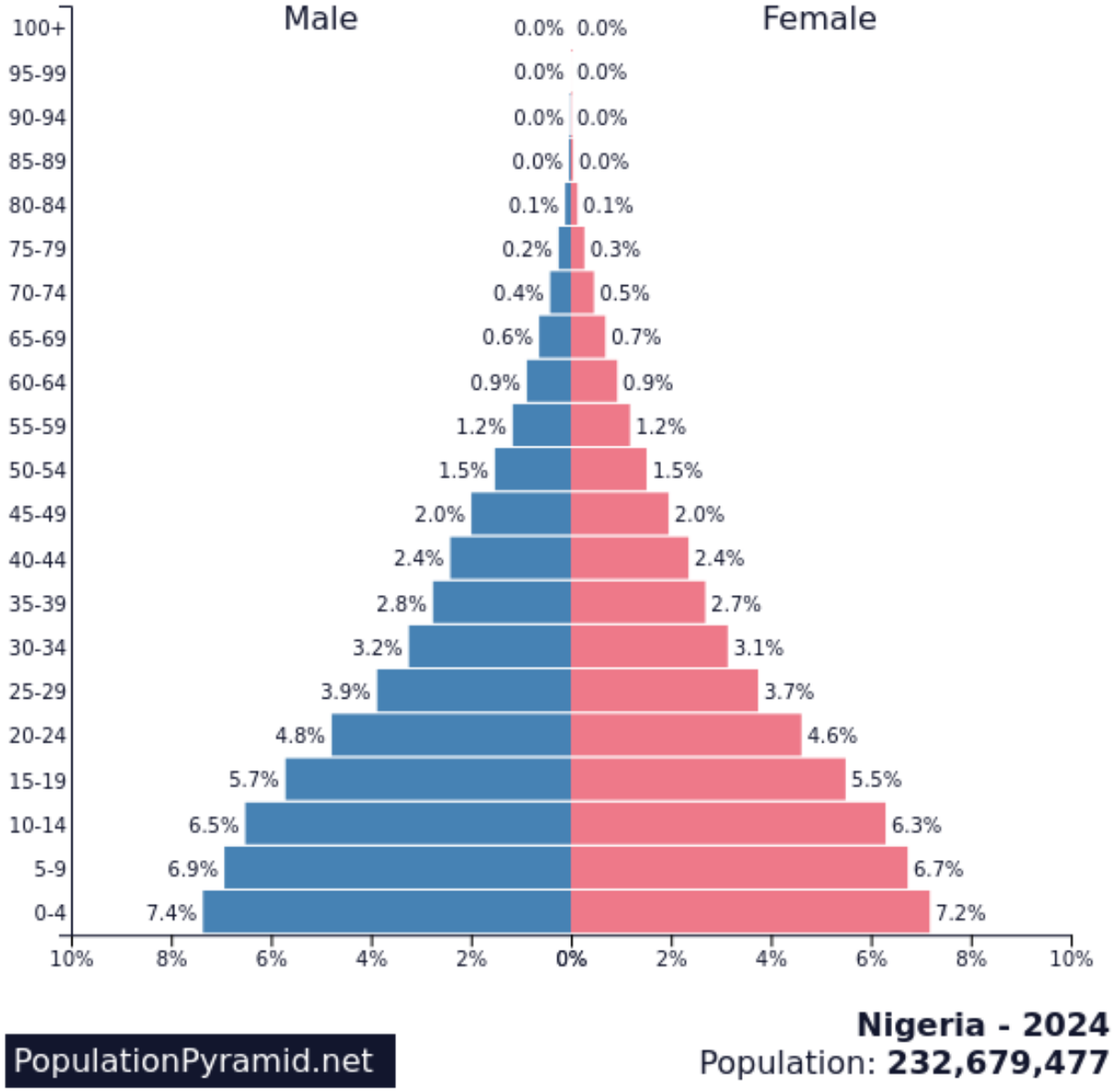
7.1 Example

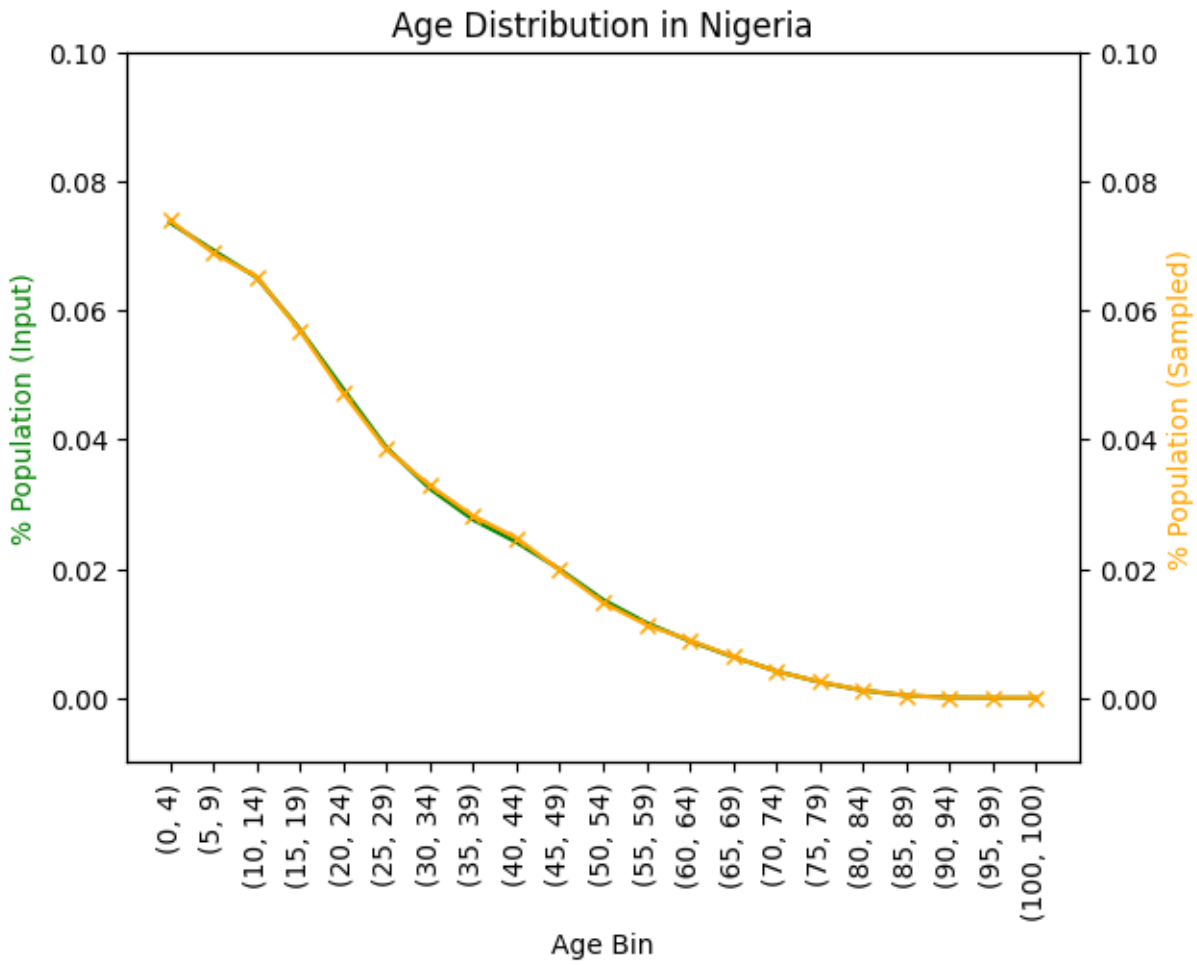
```
import numpy as np
from laser.core.demographics import load_pyramid_csv, AliasedDistribution
import importlib.util
import os

MCOL = 2
FCOL = 3

MINCOL = 0
MAXCOL = 1

# Access the bundled file dynamically
laser.core_path = importlib.util.find_spec("laser.core").origin
laser.core_dir = os.path.dirname(laser.core_path)
pyramid_file = os.path.join(laser.core_dir, "data/us-pyramid-2023.csv")

pyramid = load_pyramid_csv(pyramid_file)
sampler = AliasedDistribution(pyramid[:, MCOL])    # We'll use the male population in␣
↪this example.
n_agents = 100_000
samples = sampler.sample(n_agents)              # Sample 100,000 people from the␣
↪distribution.
# samples will be bin indices, so we need to convert them to ages.
bin_min_age_days = pyramid[:, MINCOL] * 365          # minimum age for bin, in days␣
↪(include this value)
bin_max_age_days = (pyramid[:, MAXCOL] + 1) * 365    # maximum age for bin, in days␣
↪(exclude this value)
mask = np.zeros(n_agents, dtype=bool)
ages = np.zeros(n_agents, dtype=np.int32)
for i in range(len(pyramid)):   # for each possible bin value...
    mask[:] = samples == i      # ...find the agents that belong to this bin
    # ...and assign a random age, in days, within the bin
    ages[mask] = np.random.randint(bin_min_age_days[i], bin_max_age_days[i], mask.sum())

# in some LASER models we convert current ages to dates of birth by negating the age
# dob = -ages
```

7.2 Nigeria

The population pyramid for Nigeria is available from <TBD>.

Source: https://www.populationpyramid.net/nigeria/2024/


Male
Female
100+
95-99
90-94
85-89
80-84
75-79
70-74
65-69
60-64
55-59
50-54
45-49
40-44
35-39
30-34
25-29
20-24
15-19
10-14
5-9
0-4
0.0% 0.0%
0.0% 0.0%
0.0% 0.0%
0.0% 0.0%
0.1% 0.1%
0.2% 0.3%
0.4% 0.5%
0.6% 0.7%
0.9% 0.9%
1.2% 1.2%
1.5% 1.5%
2.0% 2.0%
2.4% 2.4%
2.8% 2.7%
3.2% 3.1%
3.9% 3.7%
4.8% 4.6%
5.7% 5.5%
6.5% 6.3%
6.9% 6.7%
7.4% 7.2%
10% 8% 6% 4% 2% 0% 2% 4% 6% 8% 10%
PopulationPyramid.net
Nigeria - 2024
Population: 232,679,477


Age Distribution in Nigeria
0.10
0.08
0.06
0.04
0.02
0.00
% Population (Input)
% Population (Sampled)
(0, 4)
(5, 9)
(10, 14)
(15, 19)
(20, 24)
(25, 29)
(30, 34)
(35, 39)
(40, 44)
(45, 49)
(50, 54)
(55, 59)
(60, 64)
(65, 69)
(70, 74)
(75, 79)
(80, 84)
(85, 89)
(90, 94)
(95, 99)
(100, 100)
Age Bin

CHAPTER

EIGHT

KAPLAN-MEIER ESTIMATOR FOR PREDICTING AGE/YEAR OF DEATH

The *KaplanMeierEstimator* takes an array of cumulative deaths and returns an object that will sample from the Kaplan-Meier distribution.

A sample input array of cumulative deaths might look like this:

```
cd[0] = 687 # 687 deaths in the first year (age 0)
cd[1] = 733 # +46 deaths in the second year (age 1)
cd[2] = 767 # +34 deaths in the third year (age 2)
...
cd[100] = 100_000  # 100,000 deaths by end of year 100
```

predict_year_of_death() takes an array of current ages (in years) and returns an array of predicted years of death based on the cumulative deaths input array.

Note: predict_year_of_death() can use non-constant width age bins and will return predictions *by age bin*. In this case, it is up to the user to convert the returned bin indices to actual years.

A sample non-constant width age bin input array might look like this:

```
cd[0] = 340 # 1/2 of first year deaths in the first 3 months
cd[1] = 510 # another 1/4 (+170) of first year deaths in the next 3 months
cd[2] = 687 # another 1/4 (+177) of first year deaths in the last 6 months
cd[3] = 733 # 46 deaths in the second year (age 1)
cd[4] = 767 # 34 deaths in the third year (age 2)
...
cd[103] = 100_000  # 100,000 deaths by end of year 100
```

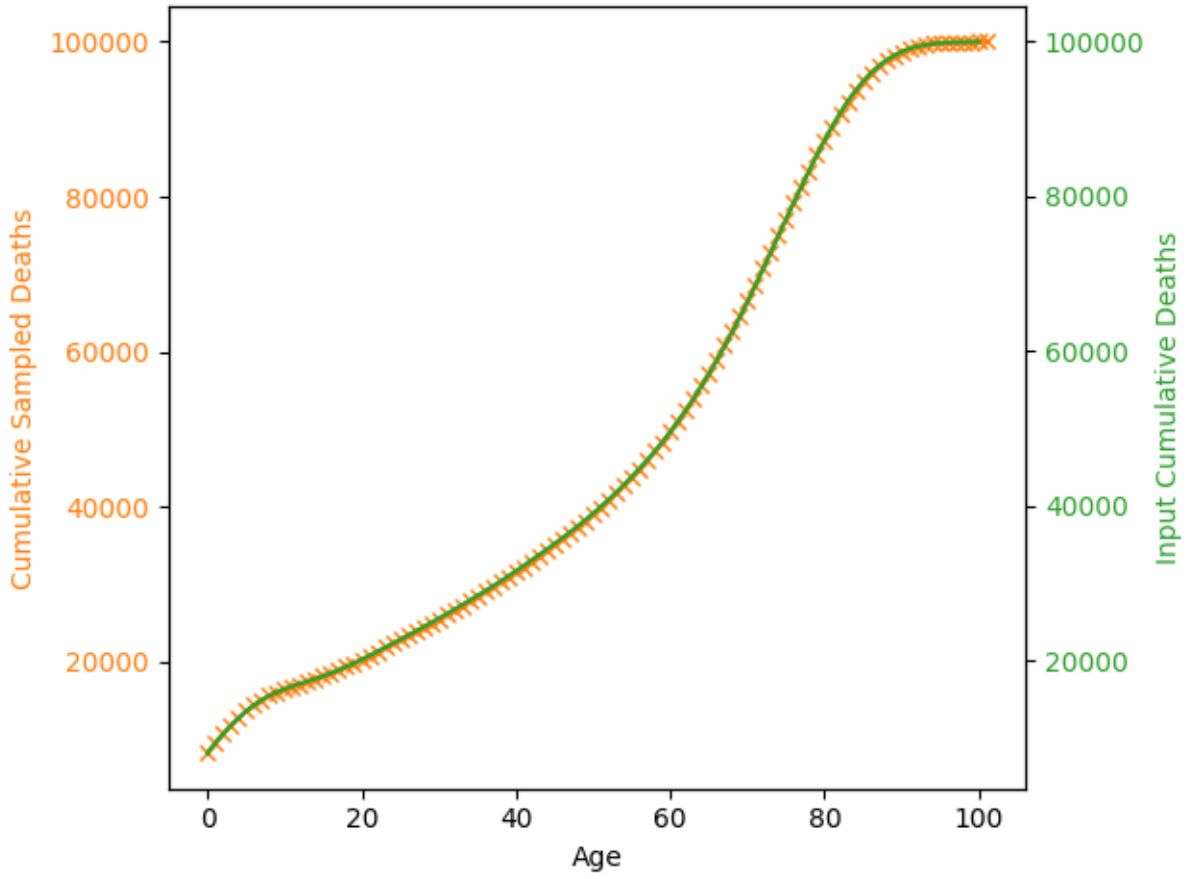
In this example, values returned from *predict_year_of_death()* would need to be mapped as follows:

```
0 -> (0, 3] months
1 -> (3, 6] months
2 -> (6, 12] months
3 -> 1 year
4 -> 2 years
...
102 -> 100 years
```

predict_age_at_death() takes an array of current ages (in days) and returns an array of predicted ages (in days) at death. The implementation assumes that the cumulative deaths input array to the estimator represents one year age bins. If you are using non-constant width age bins, you should manually convert bin indices returned from *predict_year_of_death()* to ages.

8.1 Example

```
from laser.core.demographics import KaplanMeierEstimator

estimator = KaplanMeierEstimator(cumulative)
nagents = 100_000
dobs = np.zeros(nagents)    # dates of birth, newborns = 0
dods = estimator.predict_age_at_death(dobs, max_year=100)   # dates of death in days
```



CHAPTER

NINE

SIR MODEL EXAMPLE

This document provides a comprehensive example of a single-node Susceptible-Infected-Recovered (SIR) model implemented using the *LASERFrame* and *PropertySet* libraries. This example demonstrates how to structure a disease transmission model, include components for transmission and recovery, and visualize results. The example highlights features such as reporting and plotting for model evaluation.

9.1 Introduction

The SIR model presented here simulates disease dynamics within a closed population using the *LASERFrame* framework. The population starts with a defined number of susceptible and infected individuals, progresses over time with recovery and transmission components, and tracks results for visualization. This example serves as a practical guide for modeling simple epidemic dynamics.

9.2 Code Implementation

9.2.1 Model Class

The *SIRModel* class is the core of the implementation. It initializes a population using *LaserFrame*, sets up disease state and recovery timer properties, and tracks results across timesteps.

```
import numpy as np
import matplotlib.pyplot as plt
from laser.core import LaserFrame
from laser.core import PropertySet

class SIRModel:
    def __init__(self, params):
        # Model Parameters
        self.params = params

        # Initialize the population LaserFrame
        self.population = LaserFrame(capacity=params.population_size,initial_
↪count=params.population_size)

        # Add disease state property (0 = Susceptible, 1 = Infected, 2 = Recovered)
        self.population.add_scalar_property("disease_state", dtype=np.int32, default=0)

        # Add a recovery timer property (for intrahost progression, optional for timing)
        self.population.add_scalar_property("recovery_timer", dtype=np.int32, default=0)
```

(continues on next page)

(continued from previous page)

```
        # Results tracking
        self.results = LaserFrame( capacity = 1 ) # number of nodes
        self.results.add_vector_property( "S", length=params["timesteps"], dtype=np.
↪float32 )
        self.results.add_vector_property( "I", length=params["timesteps"], dtype=np.
↪float32 )
        self.results.add_vector_property( "R", length=params["timesteps"], dtype=np.
↪float32 )

        # Components
        self.components = []

    def add_component(self, component):
        self.components.append(component)

    def track_results(self, tick):
        susceptible = (self.population.disease_state == 0).sum()
        infected = (self.population.disease_state == 1).sum()
        recovered = (self.population.disease_state == 2).sum()
        total = self.population.count
        self.results.S[tick] = susceptible / total
        self.results.I[tick] = infected / total
        self.results.R[tick] = recovered / total

    def run(self):
        for tick in range(self.params.timesteps):
            for component in self.components:
                component.step()
            self.track_results(tick)

    def plot_results(self):
        plt.figure(figsize=(10, 6))
        plt.plot(self.results.S, label="Susceptible (S)", color="blue")
        plt.plot(self.results.I, label="Infected (I)", color="red")
        plt.plot(self.results.R, label="Recovered (R)", color="green")
        plt.title("SIR Model Dynamics with LASER Components")
        plt.xlabel("Time (Timesteps)")
        plt.ylabel("Fraction of Population")
        plt.legend()
        plt.grid()
        plt.show()
        plt.savefig("gpt_sir.png")
```

9.2.2 Intrahost Progression Component

The *IntrahostProgression* class manages recovery dynamics by updating infected individuals based on a given recovery rate.

```
class IntrahostProgression:
    def __init__(self, model):
        self.population = model.population
```

(continues on next page)

(continued from previous page)

```
        # Seed the infection
        num_initial_infected = int(0.01 * model.params.population_size)  # e.g., 1%␣
↪initially infected
        infected_indices = np.random.choice(model.params.population_size, size=num_
↪initial_infected, replace=False)
        self.population.disease_state[infected_indices] = 1

        # Initialize recovery timer for initially infected individuals
        initially_infected = self.population.disease_state == 1
        self.population.recovery_timer[initially_infected] = np.random.randint(5, 15,␣
↪size=initially_infected.sum())

    def step(self):
        infected = self.population.disease_state == 1

        # Decrement recovery timer
        self.population.recovery_timer[infected] -= 1

        # Recover individuals whose recovery_timer has reached 0
        recoveries = infected & (self.population.recovery_timer <= 0)
        self.population.disease_state[recoveries] = 2
```

9.2.3 Transmission Component

The *Transmission* class manages disease spread by modeling interactions between susceptible and infected individuals.

```
class Transmission:
    def __init__(self, model):
        self.population = model.population
        self.infection_rate = model.params.infection_rate

    def step(self):
        susceptible = self.population.disease_state == 0
        infected = self.population.disease_state == 1

        num_susceptible = susceptible.sum()
        num_infected = infected.sum()
        population_size = len(self.population)

        # Fraction of infected and susceptible individuals
        fraction_infected = num_infected / population_size

        # Transmission logic: Probability of infection per susceptible individual
        infection_probability = self.infection_rate * fraction_infected

        # Apply infection probability to all susceptible individuals
        new_infections = np.random.rand(num_susceptible) < infection_probability

        # Set new infections and initialize their recovery_timer
        susceptible_indices = np.where(susceptible)[0]
        newly_infected_indices = susceptible_indices[new_infections]
        self.population.disease_state[newly_infected_indices] = 1
```

(continues on next page)

(continued from previous page)

```
        self.population.recovery_timer[newly_infected_indices] = np.random.randint(5, 15,
↪ size=newly_infected_indices.size)  # Random recovery time
```

9.2.4 Simulation Parameters

The simulation parameters are defined using the *PropertySet* class.

```
params = PropertySet({
    "population_size": 100_000,
    "infection_rate": 0.3,
    "timesteps": 160
})
```

9.2.5 Running the Simulation

The model is initialized with the defined parameters, components are added, and the simulation is run for the specified timesteps. Results are then visualized.

```
# Initialize the model
sir_model = SIRModel(params)

# Initialize and add components
sir_model.add_component(IntrahostProgression(sir_model))
sir_model.add_component(Transmission(sir_model))

# Run the simulation
sir_model.run()

# Plot results
sir_model.plot_results()
```

9.3 Conclusion

This example demonstrates a robust implementation of a single-node SIR model using *LASERFrame* and *PropertySet*. It showcases modular design, efficient result tracking, and intuitive visualization of epidemic dynamics. This example can be extended with features like vaccination or age-structured populations for advanced modeling.

CHAPTER

TEN

VITAL DYNAMICS MODEL

10.1 Overview

This module implements a vital dynamics model using the LASER framework. The model simulates population dynamics, including births and deaths, over a specified period. The key point with vital dynamics in LASER is that we don't want to add to or remove from our agent population so we preallocate inactive or preborn agents in anticipation of birthing/activating them mid-simulation, and with deaths we don't actually remove them or deallocate memory. One possible solution is a constant population scenario where every death results in a birth, so the agents are recycled in place. That has some of its own complications and is not the general case, so we are not doing that here. Key features include:

- **Births**: New agents are added to the population based on a crude birth rate (CBR).
- **Deaths**: Agents are marked as dead based on their predicted lifespan, using a Kaplan-Meier estimator.
- **Reporting**: Tracks births and deaths over time using LaserFrames.
- **Visualization**: Plots population trends over time.

10.2 Classes

The following classes are implemented:

1. **VitalDynamicsModel**: The main model class that manages the population and coordinates components. You do not have to use an enclosing Model class but it's recommended. You can call it whatever you like.
2. **BirthsComponent**: Handles births in the simulation. Can be called whatever you like, but should have a constructor and a step function. Can also have reporting and/or visualization methods.
3. **DeathsComponent**: Manages deaths in the simulation. Can be called whatever you like, but should have a constructor and a step function. Can also have reporting and/or visualization methods.

10.3 Functions

The following utility functions are provided:

1. **create_cumulative_deaths**: Generates a cumulative deaths array for use with the Kaplan-Meier estimator. In practice, you will probably provide cumulative deaths data from a file.

10.4 Sections

1. **Model Class**: - Describes and intializes the population and tracks its dynamics. Holds the main methods which manage population lifecycle.
2. **Births Component**: - Handles the addition/activation of new agents.
3. **Deaths Component**: - Handles agent removal/deactivation based on their lifespan.
4. **Utility Functions**: - Provides helper methods such as cumulative deaths generation.

10.5 Model Class

10.5.1 Code for VitalDynamicsModel

```
class VitalDynamicsModel:
    """
    Represents a vital dynamics model for simulating births and deaths in a population.

    Parameters
    ----------
    params : dict
        Dictionary containing simulation parameters:
        - `population_size`: int, initial population size.
        - `timesteps`: int, number of simulation timesteps.
        - `cbr`: float, crude birth rate per 1000 individuals per year.
    death_estimator : KaplanMeierEstimator
        Estimator used to predict lifespans based on a cumulative deaths array.
    pyramid : np.ndarray
        Population pyramid array, used to sample initial ages for the population.
    sampler : AliasedDistribution
        Distribution object for sampling initial age bins from the pyramid.

    Attributes
    ----------
    population : LaserFrame
        Frame containing properties of the population, such as `date_of_birth`, `date_of_
↪death`, and `alive`.
    report : LaserFrame
        Frame for tracking births and deaths over time.
    components : list
        List of components (e.g., `BirthsComponent`, `DeathsComponent`) added to the␣
↪model.
    """

    def __init__(self, params, death_estimator, pyramid, sampler):
        """
        Initialize the vital dynamics model and its population.
        """
        self.params = params
        # Add 1% 'fudge factor'
        capacity = int(1.01*calc_capacity(params["population_size"], params["timesteps"],
↪ params["cbr"]))
```

(continues on next page)

(continued from previous page)

```
        # Initialize the population LaserFrame
        self.population = LaserFrame(capacity=capacity, initial_count=params["population_
↪size"])
        self.population.add_scalar_property("date_of_birth", dtype=np.int32, default=-1)
        # date_of_death will be the simulation timestep where death occurs
        self.population.add_scalar_property("date_of_death", dtype=np.uint16, default=0)
        self.population.add_scalar_property("alive", dtype=np.int8, default=1)

        # Sample initial ages for the population
        n_agents = params["population_size"]
        samples = sampler.sample(n_agents)
        ages = np.zeros(n_agents, dtype=np.int32)

        for i in range(len(pyramid)):
            mask = samples == i
            ages[mask] = np.random.randint(
                pyramid[i, 0] * 365, (pyramid[i, 1] + 1) * 365, size=mask.sum()
            )

        # Set date_of_birth and predict lifespans using Kaplan-Meier estimator
        dobs = -ages # for code clarity
        self.population.date_of_birth[:n_agents] = dobs

        lifespans = death_estimator.predict_age_at_death(ages, max_year=100)
        dods = lifespans - ages # we could check that dods is non-negative to be safe
        self.population.date_of_death[:n_agents] = dods
        # Note: We could set up a PriorityQueue with the date_of_death values sorted
        # while throwing away all those which don't lie in the realm of our simulation.
        # In this implementation we will be simpler but less efficient and check all
        # dods each timestep against tick.

        # Initialize a reporting LaserFrame for births and deaths
        self.report = LaserFrame(capacity=1)
        self.report.add_vector_property("births", length=params["timesteps"], dtype=np.
↪int32)
        self.report.add_vector_property("deaths", length=params["timesteps"], dtype=np.
↪int32)

        # Components (Births and Deaths)
        self.components = []

    def add_component(self, component):
        """
        Add a simulation component to the model.

        Parameters
        ----------
        component : object
            A component such as `BirthsComponent` or `DeathsComponent`.
        """
        self.components.append(component)
```

(continues on next page)

(continued from previous page)

```
    def track_results(self, tick):
        """
        Record results from all components at the current timestep.

        Parameters
        ----------
        tick : int
            The current timestep.
        """
        for component in self.components:
            component.log(tick)

    def run(self):
        """
        Run the simulation for the specified number of timesteps.
        """
        for tick in range(self.params["timesteps"]):
            for component in self.components:
                component.step(tick)
            self.track_results(tick)

    def plot_results(self):
        """
        Visualize the births and deaths over time as a plot.
        """
        plt.figure(figsize=(10, 6))
        plt.plot(self.report.births, label="Births", color="green")
        plt.plot(self.report.deaths, label="Deaths", color="red")
        plt.title("Vital Dynamics Over Time")
        plt.xlabel("Time (Days)")
        plt.ylabel("Count")
        plt.legend()
        plt.grid()
        plt.show()
```

10.6 Births Component

10.6.1 Code for BirthsComponent

```
class BirthsComponent:
    """
    Handles births in the simulation, adding new agents to the population.

    Parameters
    ----------
    model : VitalDynamicsModel
        The vital dynamics model.
    cbr : float
        Crude birth rate per 1000 individuals per year.

    Methods
```

(continues on next page)

(continued from previous page)

```
    -------
    step(tick)
        Simulate births at the current timestep.
    log(tick)
        Record the number of births at the current timestep.
    """

    def __init__(self, model, cbr, death_estimator):
        self.population = model.population
        self.birth_rate_per_tick = cbr / (365 * 1000)
        self.report = model.report
        self.death_estimator = model.death_estimator

    def step(self, tick):
        births = int(self.birth_rate_per_tick * len(self.population))
        if births > 0:
            start, end = self.population.add(births)
            self.population.date_of_birth[start:end] = tick
            newborn_ages = np.zeros(births, dtype=np.int32)
            lifespans = self.death_estimator.predict_age_at_death(newborn_ages,max_
↪year=100)
            self.population.date_of_death[start:end] = lifespans + tick
            self.population.alive[start:end] = 1

    def log(self, tick):
        births = int(self.birth_rate_per_tick * len(self.population))
        self.report.births[tick] = births
```

10.7 Deaths Component

10.7.1 Code for DeathsComponent

```
class DeathsComponent:
    """
    Handles deaths in the simulation, marking agents as dead based on their predicted␣
↪date_of_death.

    Parameters
    ----------
    model : VitalDynamicsModel
        The vital dynamics model.
    death_estimator : KaplanMeierEstimator
        Estimator used to predict lifespans.

    Methods
    -------
    step(tick)
        Simulate deaths at the current timestep.
    log(tick)
        Record the number of deaths at the current timestep.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, model, death_estimator):
        self.population = model.population
        self.report = model.report

    def step(self, tick):
        alive = self.population.alive[:self.population.count] == 1
        dying = alive & (self.population.date_of_death[:self.population.count] <= tick)
        self.population.alive[:self.population.count][dying] = 0

    def log(self, tick):
        deaths = (self.population.alive[:self.population.count] == 0) & \
                 (self.population.date_of_death[:self.population.count] == tick)
        self.report.deaths[tick] = deaths.sum()
```

10.8 Utility Functions

10.8.1 Code for Utility Functions

```
def create_cumulative_deaths(total_population, max_age_years):
    """
    Generate a cumulative deaths array with back-loaded mortality.

    Parameters
    ----------
    total_population : int
        Total population size.
    max_age_years : int
        Maximum age in years for the cumulative deaths array.

    Returns
    -------
    cumulative_deaths : np.ndarray
        Cumulative deaths array.
    """
    ages_years = np.arange(max_age_years + 1)
    base_mortality_rate = 0.0001
    growth_factor = 2
    mortality_rates = base_mortality_rate * (growth_factor ** (ages_years / 10))
    cumulative_deaths = np.cumsum(mortality_rates * total_population).astype(int)
    return cumulative_deaths
```

Simulation Parameters

The simulation parameters are defined using the *PropertySet* class.

```
params = PropertySet({
    "population_size": 100_000,
    "cbr": 15, # Crude Birth Rate: 15 per 1000 per year
    "timesteps": 365*10 # Run for 10 years
})
```

Running the Simulation

The model is initialized with the defined parameters, components are added, and the simulation is run for the specified timesteps. Results are then visualized.

```
# Load example population pyramid
laser.core_path = importlib.util.find_spec("laser.core").origin
laser.core_dir = os.path.dirname(laser.core_path)
pyramid_file = os.path.join(laser.core_dir, "data/us-pyramid-2023.csv")
pyramid = load_pyramid_csv(pyramid_file)

# Build cumulative deaths array
sampler = AliasedDistribution(pyramid[:, 2])
cumulative_deaths = create_cumulative_deaths(params["population_size"])

# Initialize the model
model = VitalDynamicsModel(params, death_estimator, pyramid, sampler )

# Initialize and add components
model.add_component(BirthsComponent(model, params["cbr"], death_estimator))
model.add_component(DeathsComponent(model))

# Run the simulation
model.run()

# Plot results
model.plot_results()
```

10.8.2 Conclusion

The Vital Dynamics example demonstrates how to use LASER's modular components to simulate realistic population dynamics over time, including births, deaths, and age-structured demographics. By combining the `KaplanMeierEstimator` for mortality predictions with dynamic birth rates and agent-based properties, this example highlights the flexibility and scalability of the LASER framework for demographic modeling. Users can extend this baseline example with additional components, such as migration or disease dynamics, to create more complex simulations tailored to their specific research questions. This example serves as a foundational building block for models requiring detailed population structure and temporal dynamics.

CHAPTER

ELEVEN

SIMPLE SPATIAL SIR MODEL WITH SYNTHETIC DATA

This example simulates a spatial Susceptible-Infected-Recovered (SIR) model with modular components. The population is distributed across 20 nodes in a 1-D chain, with infection spreading spatially from node 0 and agents migrating based on a migration matrix.

11.1 Spatial Arrangement

The model uses a 1-D spatial structure: - **Node connections**: Each node is connected to its next neighbor in the chain, allowing migration to occur sequentially (e.g., $0 \rightarrow 1 \rightarrow 2 \ldots$). - **Infection propagation**: Infection starts in node 0 and spreads to neighboring nodes as agents migrate.

Two migration matrix options are available: 1. **Sequential Migration Matrix**: Agents can only move to their next node in the chain. 2. **Gravity Model Migration Matrix**: This provides a more realistic 2-D spatial dynamic, where migration probabilities depend on node distances and population sizes.

11.2 Population Initialization

- **Skewed Distribution**: The population is distributed across nodes using a rural-to-urban skew.
- **Timers**: Migration timers are assigned to control agent migration frequency.

11.3 Model Components

The simulation uses modular components for migration, transmission, and recovery dynamics. Each component encapsulates the logic for its specific task.

11.4 Full Code Implementation

11.4.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt
import csv
from laser.core.laserframe import LaserFrame
from laser.core.demographics.spatialpops import distribute_population_skewed as dps
from laser.core.migration import gravity
```

11.4.2 Model Class

```python
# Define the model
class MultiNodeSIRModel:
    """
    A multi-node SIR (Susceptible-Infected-Recovered) disease transmission model.

    Attributes:
        params (dict): Configuration parameters for the model.
        nodes (int): Number of nodes in the simulation.
        population (LaserFrame): Represents the population with agent-level properties.
        results (np.ndarray): Stores simulation results for S, I, and R per node.
        components (list): List of components (e.g., Migration, Transmission) added to the model.
    """

    def __init__(self, params):
        """
        Initializes the SIR model with the given parameters.

        Args:
            params (dict): Dictionary containing parameters such as population size,
                           number of nodes, timesteps, and rates for transmission/migration.
        """
        self.components = []
        self.params = params
        self.nodes = params["nodes"]
        self.population = LaserFrame(capacity=params["population_size"], initial_count=params["population_size"])

        # Define properties
        self.population.add_scalar_property("node_id", dtype=np.int32)
        self.population.add_scalar_property("disease_state", dtype=np.int32, default=0) # 0=S, 1=I, 2=R
        self.population.add_scalar_property("recovery_timer", dtype=np.int32, default=0)
        self.population.add_scalar_property("migration_timer", dtype=np.int32, default=0)

        # Initialize population distribution
        node_pops = dps(params["population_size"], self.nodes, frac_rural=0.3)
        node_ids = np.concatenate([np.full(count, i) for i, count in enumerate(node_pops)])
        np.random.shuffle(node_ids)
        self.population.node_id[:params["population_size"]] = node_ids

        # Reporting structure: Use LaserFrame for reporting
        self.results = LaserFrame( capacity=self.nodes ) # not timesteps for capacity
        for state in ["S", "I", "R"]:
            self.results.add_vector_property(state, length=params["timesteps"], dtype=np.int32)

        # Record results: reporting could actually be a component if we wanted. Or it can be
```

(continues on next page)

(continued from previous page)

```
        # done in a log/report function in the relevant component (e.g., Transmission)
        self.results.S[self.current_timestep, :] = [
            np.sum(self.population.disease_state[self.population.node_id == i] == 0)
            for i in range(self.nodes)
        ]
        self.results.I[self.current_timestep, :] = [
            np.sum(self.population.disease_state[self.population.node_id == i] == 1)
            for i in range(self.nodes)
        ]
        self.results.R[self.current_timestep, :] = [
            np.sum(self.population.disease_state[self.population.node_id == i] == 2)
            for i in range(self.nodes)
        ]

    def add_component(self, component):
        """
        Adds a component (e.g., Migration, Transmission, Recovery) to the model.

        Args:
            component: An instance of a component to be added.
        """
        self.components.append(component)

    def step(self):
        """
        Advances the simulation by one timestep, updating all components and recording␣
↪results.
        """
        for component in self.components:
            component.step()

        # Record results
        for i in range(self.nodes):
            in_node = self.population.node_id == i
            self.results[self.current_timestep, i, 0] = (self.population.disease_
↪state[in_node] == 0).sum()
            self.results[self.current_timestep, i, 1] = (self.population.disease_
↪state[in_node] == 1).sum()
            self.results[self.current_timestep, i, 2] = (self.population.disease_
↪state[in_node] == 2).sum()

    def run(self):
        """
        Runs the simulation for the configured number of timesteps.
        """
        from tqdm import tqdm
        for self.current_timestep in tqdm(range(self.params["timesteps"])):
            self.step()

    def save_results(self, filename):
        """
        Saves the simulation results to a CSV file.
```

(continues on next page)

(continued from previous page)

```
        Args:
            filename (str): Path to the output file.
        """
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Timestep", "Node", "Susceptible", "Infected", "Recovered"])
            for t in range(self.params["timesteps"]):
                for node in range(self.nodes):
                    writer.writerow([t, node, *self.results[t, node]])

    def plot_results(self):
        """
        Plots the prevalence of infected agents over time for all nodes.
        """
        plt.figure(figsize=(10, 6))
        for i in range(self.nodes):
            prevalence = self.results.I[:, i] / (
                self.results.S[:, i] +
                self.results.I[:, i] +
                self.results.R[:, i]
            )
            plt.plot(prevalence, label=f"Node {i}")

        plt.title("Prevalence Across All Nodes")
        plt.xlabel("Timesteps")
        plt.ylabel("Prevalence of Infected Agents")
        plt.legend()
        plt.show()
```

11.4.3 Migration Component Class

```
class MigrationComponent:
    """
    Handles migration behavior of agents between nodes in the model.

    Attributes:
        model (MultiNodeSIRModel): The simulation model instance.
        migration_matrix (ndarray): A matrix representing migration probabilities␣
↪between nodes.
    """

    def __init__(self, model):
        """
        Initializes the MigrationComponent.

        Args:
            model (MultiNodeSIRModel): The simulation model instance.
        """
        self.model = model

```

(continues on next page)

(continued from previous page)

```
        # Set initial migration timers
        max_timer = int(1 / model.params["migration_rate"])
        model.population.migration_timer[:] = np.random.randint(1, max_timer + 1,␣
↪size=model.params["population_size"])

        self.migration_matrix = self.get_sequential_migration_matrix(model.nodes)

        # Example customization: Disable migration from node 13 to 14
        def break_matrix_node(matrix, from_node, to_node):
            matrix[from_node][to_node] = 0
        break_matrix_node(self.migration_matrix, 13, 14)

    def get_sequential_migration_matrix(self, nodes):
        """
        Creates a migration matrix where agents can only migrate to the next sequential␣
↪node.

        Args:
            nodes (int): Number of nodes in the simulation.

        Returns:
            ndarray: A migration matrix where migration is allowed only to the next node.
        """
        migration_matrix = np.zeros((nodes, nodes))
        for i in range(nodes - 1):
            migration_matrix[i, i + 1] = 1.0
        return migration_matrix

    def step(self):
        """
        Updates the migration state of the population by determining which agents migrate
        and their destinations based on the migration matrix.
        """
        node_ids = self.model.population.node_id

        # Decrement migration timers
        self.model.population.migration_timer -= 1

        # Identify agents ready to migrate
        migrating_indices = np.where(self.model.population.migration_timer <= 0)[0]
        if migrating_indices.size == 0:
            return

        # Shuffle migrants and assign destinations based on migration matrix
        np.random.shuffle(migrating_indices)
        destinations = np.empty(len(migrating_indices), dtype=int)
        for origin in range(self.model.nodes):
            origin_mask = node_ids[migrating_indices] == origin
            num_origin_migrants = origin_mask.sum()

            if num_origin_migrants > 0:
                # Assign destinations proportionally to migration matrix
```

(continues on next page)

(continued from previous page)

```
                destination_counts = np.round(self.migration_matrix[origin] * num_origin_migrants).astype(int)
                destination_counts = np.maximum(destination_counts, 0)  # Clip negative values
                if destination_counts.sum() == 0:  # No valid destinations
                    destinations[origin_mask] = origin  # Stay in the same node
                    continue
                destination_counts[origin] += num_origin_migrants - destination_counts.sum()  # Adjust rounding errors

                # Create ordered destination assignments
                destination_indices = np.repeat(np.arange(self.model.nodes), destination_counts)
                destinations[origin_mask] = destination_indices[:num_origin_migrants]

        # Update node IDs of migrants
        node_ids[migrating_indices] = destinations

        # Reset migration timers for migrated agents
        self.model.population.migration_timer[migrating_indices] = np.random.randint(
            1, int(1 / self.model.params["migration_rate"]) + 1, size=migrating_indices.size
        )
```

11.4.4 Transmission Component Class

```
class TransmissionComponent:
    """
    Handles the disease transmission dynamics within the population.

    Attributes:
        model (MultiNodeSIRModel): The simulation model instance.
    """

    def __init__(self, model):
        """
        Initializes the TransmissionComponent and infects initial agents.

        Args:
            model (MultiNodeSIRModel): The simulation model instance.
        """
        self.model = model

    def step(self):
        """
        Simulates disease transmission for each node in the current timestep.
        """
        for i in range(self.model.nodes):
            in_node = self.model.population.node_id == i
            susceptible = in_node & (self.model.population.disease_state == 0)
            infected = in_node & (self.model.population.disease_state == 1)
```

(continues on next page)

(continued from previous page)

```
            num_susceptible = susceptible.sum()
            num_infected = infected.sum()
            total_in_node = in_node.sum()

            if total_in_node > 0 and num_infected > 0 and num_susceptible > 0:
                infectious_fraction = num_infected / total_in_node
                susceptible_fraction = num_susceptible / total_in_node

                new_infections = int(
                    self.model.params["transmission_rate"] * infectious_fraction * susceptible_fraction * total_in_node
                )

                susceptible_indices = np.where(susceptible)[0]
                newly_infected_indices = np.random.choice(susceptible_indices, size=new_infections, replace=False)

                self.model.population.disease_state[newly_infected_indices] = 1
                self.model.population.recovery_timer[newly_infected_indices] = np.random.randint(5, 15, size=new_infections)
```

11.4.5 Recovery Component Class

```
class RecoveryComponent:
    """
    Handles the recovery dynamics of infected individuals in the population.

    Attributes:
        model (MultiNodeSIRModel): The simulation model instance.
    """

    def __init__(self, model):
        """
        Initializes the RecoveryComponent.

        Args:
            model (MultiNodeSIRModel): The simulation model instance.
        """
        self.model = model

    def step(self):
        """
        Updates the recovery state of infected individuals, moving them to the recovered state
        if their recovery timer has elapsed.
        """
        infected = self.model.population.disease_state == 1
        self.model.population.recovery_timer[infected] -= 1
        self.model.population.disease_state[(infected) & (self.model.population.recovery_timer <= 0)] = 2
```

11.4.6 Run Everything

```
# Parameters
params = {
    "population_size": 1_000_000,
    "nodes": 20,
    "timesteps": 600,
    "initial_infected_fraction": 0.01,
    "transmission_rate": 0.25,
    "migration_rate": 0.001
}

# Run simulation
model = MultiNodeSIRModel(params)
model.add_component(MigrationComponent(model))
model.add_component(TransmissionComponent(model))
model.add_component(RecoveryComponent(model))
model.run()
model.save_results("simulation_results.csv")
model.plot_results()
```

11.4.7 Discussion

This simple spatial example with migration creates a set of nodes with synthetic populations and a linear connection structure. The 0th node is the 'urban' node, with the largest population, and where we seed the infection. The migration matrix just connects nodes to the next node (by index). So we expect to see infection travel sequentially from node to node. We break the connection at node 13 just to show we can.

11.5 Migration in 2 Dimensions

Now we make things a bit more interesting by taking a similar set of nodes, but creating a migration matrix from the gravtiy function, so we effectively have a 2D network with rates proportional to population sizes. Distances are still very synthetic. We change our migration function as well since we have to be a little smarter when our matrix isn't sparse.

11.5.1 Migration Component (2D)

```
class MigrationComponent2D:
    """
    Handles migration behavior of agents between nodes in the model.

    Attributes:
        model (MultiNodeSIRModel): The simulation model instance.
        migration_matrix (ndarray): A matrix representing migration probabilities␣
↪between nodes.
    """

    def __init__(self, model):
        """
        Initializes the MigrationComponent.

        Args:
```

(continues on next page)

(continued from previous page)

```
            model (MultiNodeSIRModel): The simulation model instance.
        """
        self.model = model

        # Set initial migration timers
        max_timer = int(1 / model.params["migration_rate"])
        model.population.migration_timer[:] = np.random.randint(1, max_timer + 1, size=model.params["population_size"])

        self.migration_matrix = self.get_gravity_migration_matrix(model.nodes)

    def get_gravity_migration_matrix(self, nodes):
        """
        Generates a gravity-based migration matrix based on population and distances between nodes.

        Args:
            nodes (int): Number of nodes in the simulation.

        Returns:
            ndarray: A migration matrix where each row represents probabilities of migration to other nodes.
        """
        pops = np.array([np.sum(self.model.population.node_id == i) for i in range(nodes)])
        distances = np.ones((nodes, nodes)) - np.eye(nodes)
        migration_matrix = gravity(pops, distances, k=1.0, a=0.5, b=0.5, c=2.0)
        migration_matrix = migration_matrix / migration_matrix.sum(axis=1, keepdims=True)
        return migration_matrix

    def step(self):

        """
        Executes the migration step for the agent-based model.

        This function selects a fraction of agents to migrate based on expired migration timers.
        It then changes their node_id according to the migration matrix, ensuring that movements
        follow the prescribed probability distributions.

        Steps:
        - Selects a subset of the population for migration.
        - Determines the origin nodes of migrating agents.
        - Uses a multinomial draw to assign new destinations based on the migration matrix.
        - Updates the agents' node assignments accordingly.

        Returns:
            None
        """
        # Decrement migration timers
```

(continues on next page)

(continued from previous page)

```
        self.model.population.migration_timer -= 1

        # Identify agents ready to migrate
        migrating_indices = np.where(self.model.population.migration_timer <= 0)[0]
        if migrating_indices.size == 0:
            return

        np.random.shuffle(migrating_indices)

        origins = model.population.node_id[migrating_indices]
        origin_counts = np.bincount(origins, minlength=model.params["nodes"])

        offset = 0

        for origin in range(model.params["nodes"]):
            count = origin_counts[origin]
            if count == 0:
                continue

            origin_slice = migrating_indices[offset : offset + count]
            offset += count

            row = self.migration_matrix[origin]
            row_sum = row.sum()
            if row_sum <= 0:
                continue

            fraction_row = row / row_sum
            destination_counts = np.random.multinomial(count, fraction_row)
            destinations = np.repeat(np.arange(model.params["nodes"]), destination_
↪counts)
            model.population.node_id[origin_slice] = destinations[:count]

        # Reset migration timers for migrated agents
        self.model.population.migration_timer[migrating_indices] = np.random.randint(
            1, int(1 / self.model.params["migration_rate"]) + 1, size=migrating_indices.
↪size
        )
```

11.5.2 Discussion

This example is more advanced than our first one since it moves from 1 dimension to 2, and is fully connected. We should see infection spread from the large seed node to most or potentially all of the other nodes (depending on transmission rates and migration rates and stochasticity) in a way that is broadly a function of the other nodes' populations. Though since the smaller nodes don't vary massively in population and since the model is stochastic, it will be a general correlation.

CHAPTER

TWELVE

SIMPLE SPATIAL SIR MODEL WITH REAL DATA

12.1 Design

Now we switch from synthetic population and spatial data to real population data. In this example, we have a csv file which starts off like this:

```
region_id,population,centroid_lat,centroid_long,birth_rate
Ryansoro,46482.66796875,-3.707268618580818,29.79879895068512,11.65647
Ndava,72979.296875,-3.391556716979041,29.753430749757815,15.881549
Buyengero,76468.8125,-3.8487418774123014,29.53299692786253,12.503805
Bugarama,44571.8515625,-3.6904789341549504,29.400408879716224,11.025566
Rumonge,300248.03125,-3.9622108122897663,29.45711276535364,19.567726
Burambi,63219.703125,-3.798641437985548,29.452423323952797,9.199019
Kanyosha1,115017.984375,-3.4309688424403473,29.41531324224386,37.951366
Kabezi,71913.8359375,-3.5311012728218527,29.369968675926785,31.831919
Muhuta,88141.7109375,-3.623512958448508,29.415218642943234,21.598902
```

12.1.1 Code

```
class SpatialSIRModelRealData:
    def __init__(self, params, population_data):
        """
            Initialize the mode, LASER-style, using the population_data loaded from a csv file (pandas).
            Create nodes and a migration_matrix based on populations and locations of each node.

        """
        self.params = params
        # We scale down population here from literal values but this may not be necessary.
        population_data["scaled_population"] = (population_data["population"] / params["scale_factor"]).round().astype(int)
        total_population = int(population_data["scaled_population"].sum())
        print( f"{total_population=}" )

        # Set up the properties as before
        self.population = LaserFrame(capacity=total_population, initial_count=total_population)
        self.population.add_scalar_property("node_id", dtype=np.int32)
```

(continues on next page)

(continued from previous page)

```
        self.population.add_scalar_property("disease_state", dtype=np.int32, default=0)
        self.population.add_scalar_property("recovery_timer", dtype=np.int32, default=0)
        self.population.add_scalar_property("migration_timer", dtype=np.int32, default=0)

        node_pops = population_data["scaled_population"].values
        self.params["nodes"] = len(node_pops)

        node_ids = np.concatenate([np.full(count, i) for i, count in enumerate(node_pops)])
        np.random.shuffle(node_ids)
        self.population.node_id[:total_population] = node_ids

        # seed in big node
        big_node_id = np.argmax( node_pops )
        available_population = population_data["scaled_population"][big_node_id]
        initial_infected = int(params["initial_infected_fraction"] * available_population)
        infection_indices = np.random.choice(np.where(self.population.node_id == big_node_id)[0], initial_infected, replace=False)
        self.population.disease_state[infection_indices] = 1
        self.population.recovery_timer[infection_indices] = np.random.uniform(params["recovery_time"] - 3, params["recovery_time"] + 3, size=initial_infected).astype(int)

        pop_sizes = np.array(node_pops)
        latitudes = population_data["centroid_lat"].values
        longitudes = population_data["centroid_long"].values
        distances = np.zeros((self.params["nodes"], self.params["nodes"]))

        # Nested for loop here is optimized for readbility, not performance
        for i in range(self.params["nodes"]):
            for j in range(self.params["nodes"]):
                if i != j:
                    distances[i, j] = distance(latitudes[i], longitudes[i], latitudes[j], longitudes[j])

        # Set up our migration_matrix based on gravity model and input data (pops & distances)
        self.distances = distances
        self.migration_matrix = gravity(pop_sizes, distances, k=10.0, a=1.0, b=1.0, c=1.0)
        self.migration_matrix /= self.migration_matrix.sum(axis=1, keepdims=True) # normalize
```

12.1.2 Discussion

We load the population data from the csv file, round and scale down (optional), and assign node ids. The shuffling is probably optional, but helps avoid biasing. We exploit the distance function from laser_utils (import not in code snippet).

12.2 Alternative Migration Approach

This section describes an alternative approach to modeling migration by using infection migration rather than individual movement. This method allows for computational efficiency while maintaining accurate disease transmission dynamics. Note we don't use any MigrationComponent or migration_timer in this solution.

```
import numpy as np
from laser.core.migration import gravity
from laser.core.migration import distance

class TransmissionComponent:
    """
    Transmission Component
    ======================

    This class models the transmission of disease using "infection migration"
    instead of human movement. Instead of tracking individual movement,
    infection is spread probabilistically based on a gravity-inspired network.

    This approach can significantly improve computational efficiency for
    large-scale spatial epidemic simulations.

    Attributes:
    ------------
    model : object
        The simulation model containing population and node information.
    network : ndarray
        A matrix representing the transmission probabilities between nodes.
    locations : ndarray
        Array of node latitude and longitude coordinates.
    """
    def __init__(self, model):
        """
        Initializes the transmission component by computing the infection migration␣
↪network.

        Parameters:
        -----------
        model : object
            The simulation model containing population and node information.
        """
        self.model = model
        model.nodes.add_vector_property("network", length=model.nodes.count, dtype=np.
↪float32)
        self.network = model.nodes.network

        # Extract node locations and populations from model.population_data
        self.locations = np.column_stack((model.population_data["centroid_lat"], model.
↪population_data["centroid_long"]))
        initial_populations = np.array(model.population_data["population"])

        # Initialize heterogeneous transmission factor per agent (0.5 to 2.0)
        self.model.population.tx_hetero_factor = np.random.uniform(0.5, 2.0, size=model.
```

(continues on next page)

(continued from previous page)

```
↪population.capacity)

        # Compute the infection migration network based on node populations.
        a, b, c, k = self.model.params.a, self.model.params.b, self.model.params.c, self.
↪model.params.k

        # Compute all pairwise distances in one call (this speeds up initialization␣
↪significantly)
        # from laser.core.migration import gravity, row_normalizer
        # from laser.core.migration import distance
        distances = distance(self.locations[:, 0], self.locations[:, 1])
        self.network = gravity(initial_populations, distances, k, a, b, c)
        self.network /= np.power(initial_populations.sum(), c)  # Normalize
        self.network = row_normalizer(self.network, 0.01) # 0.01=max_frac

    def step(self):
        """
        Simulates disease transmission and infection migration across the network.

        New infections are determined deterministically based on contagion levels and␣
↪susceptible fraction.
        """
        contagious_indices = np.where(self.model.population.disease_state == 1)[0]
        values = self.model.population.tx_hetero_factor[contagious_indices]  # Apply␣
↪heterogeneity factor

        # Compute contagion levels per node
        contagion = np.bincount(
            self.model.population.node_id[contagious_indices],
            weights=values,
            minlength=self.model.nodes.count
        ).astype(np.float64)

        # Apply network-based infection movement
        transfer = (contagion * self.network).round().astype(np.float64)

        # Ensure net contagion remains positive after movement
        contagion += transfer.sum(axis=1) - transfer.sum(axis=0)
        contagion = np.maximum(contagion, 0)  # Prevent negative contagion

        # Infect susceptible individuals in each node deterministically
        for i in range(self.model.nodes.count):
            node_population = np.where(self.model.population.node_id == i)[0]
            susceptible = node_population[self.model.population.disease_state[node_
↪population] == 0]

            if len(susceptible) > 0:
                # Compute new infections deterministically based on prevalence and␣
↪susceptible fraction
                num_new_infections = int(min(len(susceptible), (
                    self.model.params.transmission_rate * contagion[i] *␣
↪len(susceptible) / len(node_population)
```

(continues on next page)

(continued from previous page)

```
                )))

                # Randomly select susceptible individuals for infection
                new_infected_indices = np.random.choice(susceptible, size=num_new_
↪infections, replace=False)
                self.model.population.disease_state[new_infected_indices] = 1

                # Assign recovery timers to newly infected individuals
                self.model.population.recovery_timer[new_infected_indices] = np.random.
↪randint(5, 15, size=num_new_infections)

        # TODO: Potential Performance Improvement: Consider using a sparse␣
↪representation for `network`
        # if many connections have very low probability. This would speed up matrix␣
↪multiplications significantly.

        # TODO: Investigate parallelization of contagion computation for large-scale␣
↪simulations
        # using Numba or JIT compilation to optimize the loop structure.
```

CHAPTER

THIRTEEN

POPULATION INITIALIZATION, SQUASHING, AND SNAPSHOT MANAGEMENT IN LASER

As the number agents in your LASER population model grows (e.g., 1e8), it can become computationally expensive and unnecessary to repeatedly run the same (sophisticated) initialization routine every sim. In many cases -— particularly during model calibration -— it is far more efficient to initialize the population once, save it, and then reload the initialized state for subsequent runs.

This approach is especially useful when working with EULAs – Epidemiologically Uninteresting Light Agents. For example it can be a very powerful optimization to compress all the agents who are already (permanently) recovered or immune in a measles or polio model into a number/bucket. In such models, the majority of the initial population may be in the "Recovered" state, potentially comprising 90% or more of all agents. If you are simulating 100 million agents, storing all of them can result in punitive memory usage.

To address this, LASER supports a **squashing** process. Squashing involves *defragmenting the data frame* such that all epidemiologically active or "interesting" agents (e.g., Susceptible or Infectious) are moved to the beginning of the array or table, and less relevant agents (e.g., Recovered) are moved to the end. Though please note that you should assume that squashed agent data is overwritten.

Some notes about squashing:

- The population count is adjusted so that all *for* loops and step functions iterate **only over the active population**.
- This not only reduces memory usage but also improves performance by avoiding unnecessary computation over inactive agents.

Some caveats about using saved populations: - You will want to be confident that the saved population is sufficiently randomized and representative; - If you are calibrating parameters used to create the initial population in the first place, you'll need to recreate those parts of the population after loading, diminishing the benefit of the save/load approach.

When saving a **Snapshot**, note that only the active (unsquashed) portion of the population is saved. Upon reloading:

- Only this subset is allocated in memory.
- This prevents the performance penalty of managing large volumes of unused agent data.

13.1 Important Detail

Before squashing, you should **count and record** the number of recovered (or otherwise squashed) agents. This count should be stored in a summary variable —- typically the `R` column of the results data frame. This ensures your model retains a complete epidemiological record even though the agents themselves are no longer instantiated.

CHAPTER

FOURTEEN

IMPLEMENTATION DEATILS: HOW TO ADD SQUASHING, SAVING, LOADING, AND CORRECT R TRACKING TO A LASER SIR MODEL

14.1 1. Add Squashing

- **Add a `squash_recovered()` function** This should call *LaserFrame.squash(...)* with a boolean mask that includes non-recovered agents (*disease_state != 2*). You may choose a different criterion, such as age-based squashing.
- **Count your "squashed away" agents first** You must compute and store all statistics related to agents being squashed *before* the *squash()* call. After squashing, only the left-hand portion of the arrays (up to *.count*) remains valid.
- **Seed infections after squashing** If your model seeds new infections (*disease_state == 1*), this must happen *after* squashing. Otherwise, infected agents may be inadvertently removed.
- **Store the squashed-away totals by node** Before squashing, compute and record node-wise totals (e.g., recovered counts) in *results.R[0, :]* so this pre-squash information persists.
- **Optionally simulate EULA effects once and save** If modeling aging or death among squashed agents, simulate this up front and store the full *[time, node]* matrix (e.g., *results.R[:, :]*). This avoids recomputation at runtime.

14.2 2. Save Function

Implement a `save(path)` method:

- Use `LaserFrame.save_snapshot(path, results_r=..., pars=...)`
- Include:
  - The squashed population (active agents only)
  - The `results.R` matrix containing both pre-squash and live simulation values
  - The full parameter set in a `PropertySet`

14.3 3. Load Function

Implement a `load(path)` class method:

- Call `LaserFrame.load_snapshot(path)` to retrieve:
  - Population frame
  - Results matrix
  - Parameters

- Set `.capacity = .count` if not doing births, else set capacity based on projected population growth from count.
- Reconstruct all components using `init_from_file()`

> **Warning**
>
> **Vital Dynamics Considerations**
>
> When modeling vital dynamics—especially births—there is an additional step to ensure consistency after loading:
>
> - **Property initialization for unborn individuals** must be repeated if your model pre-assigns properties up to `.capacity`. For example, if timers or demographic attributes (like `date_of_birth`) are pre-initialized at `t=0`, you must ensure this initialization is re-applied after loading, because only the `.count` population is reloaded, not the future `.capacity`.
>
> Failing to do so may result in improperly initialized agents being birthed after the snapshot load, which can lead to subtle or catastrophic model errors.

14.4 4. Preserve EULA'd Results

- **Use ``+=`` to track new recoveries alongside pre-squash R values** In `run()`, use additive updates so that pre-saved recovered agents are preserved:

```
self.results.R[t, nid] += ((self.population.node_id == nid) &
                           (self.population.disease_state == 2)).sum()
```

This ensures your output accounts for both squashed-away immunity and recoveries during the live simulation.

CHAPTER

FIFTEEN

COMPLETE SIR LASER MODEL WITH SQUASHING AND SNAPSHOT SUPPORT

This example demonstrates a complete SIR model using LASER, featuring:

- Agent squashing based on recovery state
- Pre-squash result capture
- Snapshot saving and loading
- Node-level time series tracking
- Plotting of total S, I, and R dynamics

```
import numpy as np
import click
import matplotlib.pyplot as plt
from pathlib import Path

from laser.core import LaserFrame, PropertySet

class Transmission:
    """
    A simple transmission component that spreads infection within each node.
    """
    def __init__(self, population, pars):
        self.population = population
        self.pars = pars

    def step(self):
    """
    For each node in the population, calculate the number of new infections as a␣
↪function of:
    - the number of infected individuals,
    - the number of susceptibles,
    - adjustments for migration and seasonality,
    - and individual-level heterogeneity.

    Then, select new infections at random from among the susceptible individuals in each␣
↪node,
    and initiate infection in those individuals.
    """
    pass  # Implementation omitted for documentation purposes
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def init_from_file(cls, population, pars):
        return cls(population, pars)

class Progression:
    """
    A simple progression component that recovers infected individuals probabilistically.
    """
    def __init__(self, population, pars):
        self.population = population
        self.pars = pars

    def step(self):
    """
    At each time step, update the disease state of infected individuals based on the␣
↪model's
    progression logic. This may be driven by probabilities, timers, or other intrahost␣
↪dynamics.
    """
    pass  # Implementation omitted for documentation

    @classmethod
    def init_from_file(cls, population, pars):
        return cls(population, pars)

class RecoveredSquashModel:
    """
    A simple multi-node SIR model demonstrating use of LASER's squash and snapshot␣
↪mechanisms.
    """
    def __init__(self, num_agents=100000, num_nodes=20, timesteps=365):
        self.num_agents = num_agents
        self.num_nodes = num_nodes
        self.timesteps = timesteps
        self.population = LaserFrame(capacity=num_agents, initial_count=num_agents)
        self.population.add_scalar_property("node_id", dtype=np.int32)
        self.population.add_scalar_property("disease_state", dtype=np.int8)  # 0=S, 1=I,␣
↪2=R

        self.results = LaserFrame(capacity=self.num_nodes)
        self.results.add_vector_property("S", length=timesteps, dtype=np.int32)
        self.results.add_vector_property("I", length=timesteps, dtype=np.int32)
        self.results.add_vector_property("R", length=timesteps, dtype=np.int32)

        self.pars = PropertySet({
            "r0": 2.5,
            "migration_k": 0.1,
            "seasonal_factor": 0.8,
            "transmission_prob": 0.2,
            "recovery_days": 14
        })
```

(continues on next page)

(continued from previous page)

```
        self.components = [
            Transmission(self.population, self.pars),
            Progression(self.population, self.pars)
            # could add other components like vaccination
        ]

    def initialize(self):
        np.random.seed(42)
        self.population.node_id[:] = np.random.randint(0, self.num_nodes, size=self.num_
↪agents)
        recovered = np.random.rand(self.num_agents) < 0.6
        self.population.disease_state[:] = np.where(recovered, 2, 0)

    def seed_infections(self):
        susceptible = self.population.disease_state == 0
        num_seed = max(1, int(0.001 * self.population.count))
        seed_indices = np.random.choice(np.where(susceptible)[0], size=num_seed,␣
↪replace=False)
        self.population.disease_state[seed_indices] = 1

    def squash_recovered(self):
        """
        Removes all agents who are recovered (state 2).
        This reduces memory footprint and speeds up simulation.
        """
        keep = self.population.disease_state[:self.population.count] != 2
        self.population.squash(keep)

    def populate_results(self):
        """
        Populate initial R values before squashing to reflect the pre-squash immunity␣
↪landscape.
        """
        for nid in range(self.num_nodes):
            initial_r = ((self.population.disease_state == 2) & (self.population.node_id␣
↪== nid)).sum()
            decay = np.linspace(initial_r, initial_r * 0.9, self.timesteps, dtype=int)
            self.results.R[:, nid] = decay
        print("Initial R counts per node:", self.results.R[0, :])
        print("Total initial R (summed):", self.results.R[0, :].sum())

    def run(self):
        for t in range(self.timesteps):
            for component in self.components:
                component.step()
            for nid in range(self.num_nodes):
                self.results.S[t, nid] = ((self.population.node_id == nid) & (self.
↪population.disease_state == 0)).sum()
                self.results.I[t, nid] = ((self.population.node_id == nid) & (self.
↪population.disease_state == 1)).sum()
                self.results.R[t, nid] += ((self.population.node_id == nid) & (self.
```

(continues on next page)

(continued from previous page)

```
↪population.disease_state == 2)).sum()

    def save(self, path):
        """
        Save the current model state to an HDF5 file, including population frame,
        pre-squash results, and simulation parameters.
        """
        self.population.save_snapshot(path, results_r=self.results.R, pars=self.pars)

    @classmethod
    def load(cls, path):
        """
        Reload a model from an HDF5 snapshot. Note: reloaded population will have
        only post-squash agents (e.g., susceptibles and infected).
        """
        pop, results_r, pars = LaserFrame.load_snapshot(path)
        model = cls(num_agents=pop.capacity, num_nodes=results_r.shape[1],␣
↪timesteps=results_r.shape[0])
        model.population = pop
        model.results.R[:, :] = results_r
        model.pars = PropertySet(pars)
        model.pars["transmission_prob"] /= 10  # example modification after reload
        model.components = [
            Transmission.init_from_file(model.population, model.pars),
            Progression.init_from_file(model.population, model.pars)
        ]
        return model

    def plot(self):
        """
        Plot the time series of total S, I, and R across all nodes.
        """
        # details omitted

@click.command()
@click.option("--init-pop-file", type=click.Path(), default=None, help="Path to snapshot␣
↪to resume from.")
@click.option("--output", type=click.Path(), default="model_output.h5")
def main(init_pop_file, output):
    if init_pop_file:
        model = RecoveredSquashModel.load(init_pop_file)
        model.run()
        model.plot()
    else:
        model = RecoveredSquashModel()
        model.initialize()
        model.seed_infections()
        model.populate_results()
        model.squash_recovered()
        model.save(output)
        print(f"Initial population saved to {output}")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

CHAPTER

SIXTEEN

LASER PERFORMANCE OPTIMIZATION

It's fairly easy to generate working LASER model code that uses NumPy and performs well for populations of 100,000 agents. We expect LASER code to perform efficiently for **200,000,000 agents** (or more). The challenge is how to get there.

16.1 Identifying Performance Bottlenecks

Typically, we do **not** recommend running the code through a profiler—at least not initially. Instead, we take advantage of **LASER's highly modular structure** and **AI-driven optimization**.

The first step is to **add simple timing code** that tracks the total time spent in each component over a realistic simulation. Then, **plot a pie chart** at the end to visualize where the most time is spent. A simple way to track execution time is using the `time` package. Example code:

```
def run(self):
    self.component_times = {component.__class__.__name__: 0 for component in self.
↪components}
    self.component_times["reporting"] = 0
    for tick in tqdm(range(self.pars.timesteps)):
        for component in self.components:
            start_time = time.time()  # Start timing for the component
            component.step()
            end_time = time.time()  # End timing for the component

            # Accumulate the time taken for this component
            elapsed_time = end_time - start_time
            component_name = component.__class__.__name__
```

This often reveals **the top 1 to 3 performance bottlenecks**. Focus first on the biggest offender—it provides the most opportunity for speedup. Often, the largest bottleneck is **not** what you might instinctively expect. Avoid optimizing a component only to find out it contributes just **a small percentage of the total runtime**. A modest improvement in the runtime of an "expensive" component is often more effective than spending a lot of time on highly optimizing a component which only accounts for a small fraction of runtime. Also, make sure that your reporting code is being measured and reported, ideally in its own 'bucket'. This may be easier or harder depending on how you are doing reporting. Since reporting usually involves counting over the entire population, it usually shows up as a hotspot sooner or later. Fortunately, it's usually fairly easy to speed up. Or even eliminate.

16.2 Leveraging AI for Code Optimization

Once you've identified the **slowest component**, the easiest way to improve performance is by using **ChatGPT**. Try prompting with:

```
"This code is much too slow. (My arrays are all about 1e6 or 1e7 in size.)"
```

If your code consists mainly of **for-loops** without much **NumPy**, you can add:

```
"Is there anything we can vectorize better with NumPy?"
```

This approach can often **transform a naive implementation into a highly optimized one**.

16.3 Unit Tests for Performance and Accuracy

Instead of testing performance within the full simulation, consider building **unit tests**. This ensures **correctness** while optimizing for **speed**.

- Use AI to generate **unit tests** that validate output against a known correct (but slower) version.
- Include **performance benchmarks** in the tests.
- Ensure **large array sizes** (e.g., **1 million+ elements**) to get meaningful speed comparisons.

16.4 Optimizing with NumPy and Numba

After achieving good performance with **NumPy**, consider trying **Numba** for further improvements.

Even if you're new to Numba, **ChatGPT** can generate optimized solutions easily. Keep in mind:

- **Numba moves back to explicit for-loops** (unlike NumPy, which uses vectorization syntax).
- GPT's first solution may use `range` instead of `prange`. Prompt it with:

  ```
  "Can we parallelize this with prange?"
  ```

- If your code involves **common counters**, **atomic operations** may become a bottleneck. Ask GPT about:

  ```
  "Can we use thread-local storage to avoid atomic operations?"
  ```

- **Numba may be slower than NumPy for small arrays** (e.g., thousands or tens of thousands of elements). Test with **at least 1 million elements**.

16.4.1 Further Tricks

- Don't duplicate: Sometimes reporting will duplicate transmission code and need to be combined.
- Never append. There may be cases where you are collecting information as it happens without knowing ahead of time how many rows/entries/elements you'll need. This is easy in Python using list appending, for example, but that's a performance killer. Really try to find a way to figure out ahead of time how many entries there will be, and then allocate memory for that, and insert into the existing row.
- Some components have long time-scales, like mortality. By default you are probably going to end up doing most component steps every timestep. You can probably get away with doing mortality updates, for example, far less often. You can experiement with weekly, fortnightly or monthly updates, depending on the timescale of the component you're optimizing. Just be sure to move everything forward by a week if you're only doing the update every week. And expect "blocky" plots. Note that there are fancier solutions like 'strided sharding' (details omitted).

When **prompting AI**, use **questions rather than directives**. Example:

```
"Do you think it might be better to...?"
```

This prevents oversteering the AI into suboptimal solutions.

16.5 Beyond Numba: C and OpenMP

If the best **Numba** solution still isn't fast enough, consider **compiled C**.

- Use **ctypes** to call C functions from Python.
- Mention **"use OpenMP"** in AI prompts if parallelization is possible.
- Ask:

  ```
  "Can you generate an OpenMP solution with the best pragmas?"
  ```

- The more CPU cores available, the **greater the potential speedup**. That said, it's usually a case of diminishing returns as one goes from 8 cores to 16 and to 32. Our research shows that often you're better off running 4 sims across 8 cores each than running 1 sim on all 32 cores available. Also be aware that with both Numba and OpenMP you can constrain the number of cores used to less than the number available by setting the appropriate environment variable. (Numba environment variable = NUMBA_NUM_THREADS; OpenMP environment variable = OMP_NUM_THREADS)

16.5.1 Advanced Hardware-Dependent Performance Improvements

Beyond compiled C extensions using OpenMP for parallelization across CPU cores, even greater performance gains can be achieved by leveraging hardware-specific optimizations.

16.6 SIMD

Modern CPUs include low-level **Single Instruction, Multiple Data (SIMD)** instruction sets that enable direct access to vectorized operations. While compilers attempt to generate optimal SIMD instructions automatically, they are not always perfect.

In theory, writing SIMD code manually can lead to significant performance gains, but this requires deep, architecture-specific knowledge. Fortunately, AI-assisted development tools can help generate such code. However, in practice, achieving meaningful speedups for complex use cases remains challenging. Additionally, since SIMD instruction sets vary by hardware, code optimized for a development machine may not work on a different target machine. Consult a developer to determine applicability for your use case.

16.7 GPU

GPUs can provide massive speedups when used effectively, but several challenges must be considered:

- GPU hardware must be available on the target machine.
- GPU-specific code needs to be written, often using CUDA (for NVIDIA GPUs) or other frameworks like OpenCL or ROCm.
- The overhead of transferring data between CPU and GPU memory can negate performance benefits unless the system has unified memory.

We continue to explore GPU acceleration for LASER, particularly for cases where computational workloads justify the overhead of GPU execution.

16.8 Final Thoughts

In some cases, an algorithm may be **inherently sequential**, meaning **parallelization won't help**. Be mindful that AI might not always indicate when you're **hitting a fundamental limitation**.

By following this process—**profiling via timing, leveraging AI, and incrementally optimizing with NumPy, Numba, and C**—you can take LASER models from **functional** to **high-performance** at massive scales.

CHAPTER

SEVENTEEN

CALIBRATION WORKFLOW FOR LASER MODELS

This guide explains how to calibrate a LASER model using Optuna. Calibration is the process of adjusting model parameters (e.g., transmission rate, R0) so that simulation outputs match reference data (e.g., case counts, prevalence curves). This document assumes you've already built and tested a working LASER model.

17.1 Prerequisites

- A functioning, tested LASER model.
- Python environment with *laser-core*, *optuna*, *pandas*, and *numpy* installed.
- (Optional) Docker Desktop installed if running distributed calibration.

17.2 Simple Local Calibration

1. **Expose Parameters in Your Model** Ensure your LASER model can load and apply parameters you wish to calibrate. These are typically passed through a *params* dictionary or a *PropertySet* and might include:
   - Basic reproduction number (R0)
   - Duration of infection
   - Seeding prevalence
2. **Write Post-Processing Code** Modify your model to save key outputs (e.g., number of infected individuals over time) to a CSV file. For example, use:

   ```
   save_results_to_csv(sim.results)
   ```

   This CSV will be used later by the objective function.
3. **Create the Objective Function** Write a Python script, usually named *objective.py*, containing a function like this:

   ```
   def objective(trial):
       # Load trial parameters
       R0 = trial.suggest_float("R0", 1.0, 3.5)

       # Run model (via subprocess, or function call)
       run_model(R0)

       # Load model output and reference data
       model_df = pd.read_csv("output.csv")
       ref_df = pd.read_csv("reference.csv")
   ```

(continues on next page)

(continued from previous page)

```
    # Compare and return score
    error = np.mean((model_df["I"] - ref_df["I"])**2)
    return error
```

Tip: You can write unit tests for your objective function by mocking model outputs.

4. **Test Objective Function Standalone** Before integrating with Optuna, run your objective function directly to ensure it works:

```
from objective import objective
from optuna.trial import FixedTrial

score = objective(FixedTrial({"R0": 2.5}))
print(f"Test score: {score}")
```

Expected Result: A numeric score. If it crashes, check CSV paths and data types.

5. **Run Simple Calibration (SQLite, No Docker)** Use the *calib/worker.py* helper to run a local test study with a small number of trials.

Linux/macOS (Bash or similar):

```
export STORAGE_URL=sqlite:///example.db && python3 calib/worker.py --num-trials=10
```

Windows (PowerShell):

```
$env:STORAGE_URL="sqlite:///example.db"; python calib/worker.py --num-trials=10
```

This is helpful for debugging. Consider running a scaled-down version of your model to save time.

17.3 Local Dockerized Calibration

6. **Dockerize Your Model and Objective** Use the provided *Dockerfile* to build a container that includes both your model and objective function. Do this from the main directory.

```
docker build . -f calib/Dockerfile -t idm-docker-staging.packages.idmod.org/laser/
↪laser-polio:latest
```

7. **Create Docker Network** You'll need a shared network so your workers and database container can communicate:

```
docker network create optuna-network
```

8. **Launch MySQL Database Container**

```
docker run -d --name optuna-mysql --network optuna-network -p 3306:3306 \
  -e MYSQL_ALLOW_EMPTY_PASSWORD=yes \
  -e MYSQL_DATABASE=optuna_db mysql:latest
```

9. **Launch Calibration Worker**

```
docker run --rm --name calib_worker --network optuna-network \
  -e STORAGE_URL="mysql://root@optuna-mysql:3306/optuna_db" \
  idm-docker-staging.packages.idmod.org/laser/laser-polio:latest \
  --study-name test_polio_calib --num-trials 1
```

If that works, you can change the study name or number of trials.

Troubleshooting: If this fails, try running the worker interactively and debug inside:

```
docker run -it --network optuna-network --entrypoint /bin/bash idm-docker-
↪staging.packages.idmod.org/laser/laser-polio:latest
```

10. **Monitor Calibration Progress**

Use Optuna CLI. You should be able to pip install optuna.

```
optuna trials \
  --study-name=test_polio_calib \
  --storage "mysql+pymysql://root:@localhost:3306/optuna_db"

optuna best-trial \
  --study-name=test_polio_calib \
  --storage "mysql+pymysql://root:@localhost:3306/optuna_db"
```

17.4 Cloud Calibration

11. **Push Docker Image to Registry**

If you've built a new docker image, you'll want to push it so it's available to AKS.

```
docker push idm-docker-staging.packages.idmod.org/laser/laser-polio:latest
```

12. **Cloud Deployment**

This step assumes you have secured access to an Azure Kubernetes Service (AKS) cluster. You may need to obtain or generate a new kube config file. Detailed instructions for that are not included here. This step assumes the cluster corresponding to your config is up and accessible.

```
cd calib/cloud
```

- Edit config file. Edit *cloud_calib_config.py* to set the storage_url to:

```
"mysql+pymysql://optuna:superSecretPassword@localhost:3306/optunaDatabase"
```

And set the study name and number of trials per your preference. Detailed documentation of the other parameters is not included here.

- Launch multiple workers:

```
python3 run_calib_workers.py
```

13. **View Final Results**

- Forward port to local machine. Note that is the first instruction to rely on installing *kubectl*. Open a bash shell if necessary.

```
kubectl port-forward mysql-0 3306:3306 &
```

- Use Optuna CLI to check results:

```
optuna trials \
  --study-name=test_polio_calib \
  --storage "mysql+pymysql://optuna:superSecretPassword@localhost:3306/
↪optunaDatabase"

optuna best-trial \
  --study-name=test_polio_calib \
  --storage "mysql+pymysql://optuna:superSecretPassword@localhost:3306/
↪optunaDatabase"
```

- Generate a report on disk about the study (can be run during study or at end).

```
python3 report_calib_aks.py
```

- Launch Optuna Dashboard

```
python -c "import optuna_dashboard; optuna_dashboard.run_server('mysql+pymysql:/
↪/optuna:superSecretPassword@127.0.0.1:3306/optunaDatabase')"
```

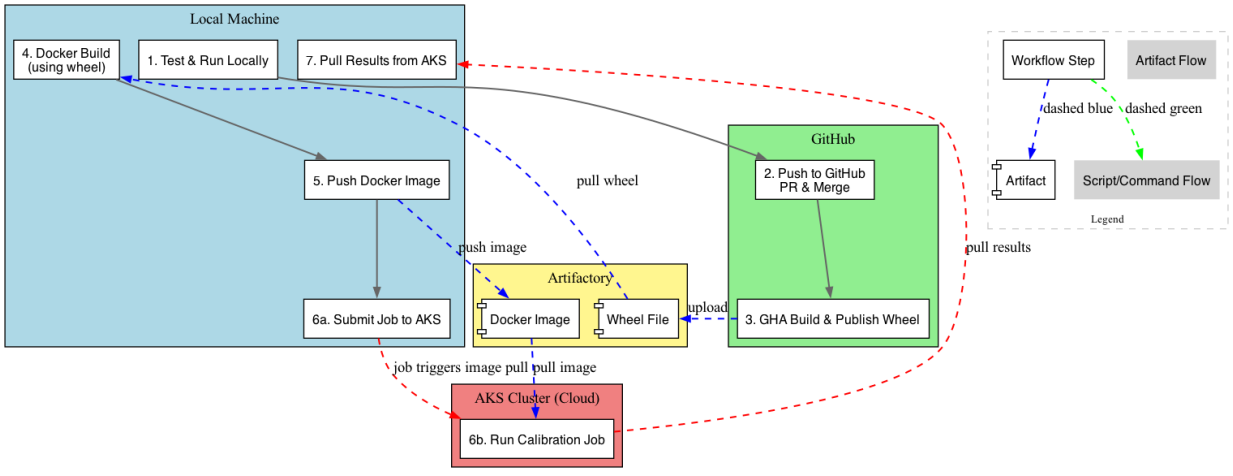
17.5 Expected Output

- A best-fit parameter set (*R0*, etc.) that minimizes error.
- An Optuna study saved in MySQL or SQLite.
- Log files or CSVs showing score over time.

17.6 Error Handling

- Missing CSVs: Ensure output files are written by the model before scoring.
- Model crashes: Check Docker logs (*docker logs <container>*) or run interactively.
- Database connection errors: Confirm Docker network and container health. Ensure MySQL is listening on the expected port.

17.7 Iterative Development Cycle



17.7.1 Workflow Steps

1. **Test & Run Locally**

   Test your model and calibration code locally before committing or containerizing.

   ```
   python3 calibrate.py --study-name=test_local --num-trials=3
   ```

2. **Push to GitHub**

   Push your changes to GitHub and submit a pull request for review. Once approved, merge to the default branch (e.g. *main* or *develop*).

3. **GHA Build & Publish Wheel**

   GitHub Actions (GHA) will automatically build a Python wheel and publish it to your private PyPI/Artifactory.

4. **Docker Build (using wheel)**

   On your local machine, build a Docker image using the freshly published wheel.

   ```
   docker build -t idm-docker-staging.packages.idmod.org/laser/laser-
   ↪polio:latest .
   ```

5. **Push Docker Image**

   Push the built image to your container registry so it's accessible from AKS.

   ```
   docker push idm-docker-staging.packages.idmod.org/laser/laser-polio:latest
   ```

6. **Submit and Run Calibration Job**

   a. **Submit Job to AKS**:

   Launch your Kubernetes job to run calibration using the new image.

   ```
   python3 run_calib_workers.py
   ```

   b. **Run Calibration Job**:

   The cluster pulls the image and executes the calibration job according to the job spec.

7. **Pull Results from AKS**

   See above.

17.7.2 Notes

- If port 8080 is already in use when launching the dashboard, use *port=8081* or another free port.
- Make sure your port-forwarding process is active whenever running Optuna CLI or dashboard from your local machine.
- Each iteration through this workflow can test new parameters, updated logic, or bug fixes — without affecting production deployments.

17.8 Next Steps

Once you've completed calibration: - Analyze the best-fit parameters. - Re-run your model using the optimal settings. - Generate plots or reports to summarize calibration quality.

CHAPTER

EIGHTEEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

18.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

18.2 Documentation improvements

LASER could always use more documentation, whether as part of the official LASER docs, in docstrings, or even on the web in blog posts, articles, and such.

18.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/InstituteforDiseaseModeling/laser/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

18.4 Development

To set up *laser* for local development:

1. Fork laser (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/laser.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

18.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

18.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

NINETEEN

AUTHORS

- Christopher Lorton - https://www.idmod.org
- Jonathan Bloedow - https://www.idmod.org
- Katherine Rosenfeld - https://www.idmod.org
- Kevin McCarthy - https://www.idmod.org

CHAPTER

TWENTY

CHANGELOG

20.1 0.0.1 (2023-11-18)

- First release on PyPI.

CHAPTER

TWENTYONE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

G

I

K

L

M

N

P

R

S

T

U

W