Poliosim

Release 0.10.10

Institute for Disease Modeling

Dec 02, 2021

CONTENTS

Poliosim is a stochastic agent-based simulator, written in Python, for exploring and analyzing the polio epidemic.

There's a lot here, where should I start?

- Take a quick look at the overview, which provides a general introduction.
- If you're looking for a specific feature or keyword, you should be able to find it with the search feature (top left).
- Still have questions? Send us an email at feedback@idmod.org. We're happy to help!

CHAPTER

ONE

FULL CONTENTS

1.1 Poliosim overview

The repo is organized as follows:

- Poliosim, in the folder `poliosim`, is a standalone Python library for performing polio simulations.
- Scripts that use Poliosim to run analyses are in the `scripts`, `examples`, and `pipeline` folder.
- Poliosim tests are in the `tests` folder.

1.1.1 Installation

Run `pip install -e .` to install Poliosim and its required dependencies. This will make `poliosim` available on the Python path.

1.1.2 Usage

You can run a simple example with `tests/test_simple.py`. For more advanced usage, see the scripts folders listed above, e.g. `pipeline/example_drone.py`.

1.1.3 Documentation

Documentation is available at https://docs.idmod.org/projects/poliosim/en/latest/.

1.1.4 Contributing

1.1.4.1 Before starting

- Everything you're working on must be linked to an issue. If you notice that something needs to be done (even small things or things nearly finished) and there isn't an issue for it, create an issue. This helps track who is doing what and why.
- ALL PRs should be linked to at least one issue. If you find yourself working on a PR and there's no issue associated with it, please check with the project owner before proceeding.
- All PRs will be assigned to the project owner for review automatically, and typically you would also assign to at least one other person for review, depending on relevance.
- At times there may be a backlog of issues, but there should never be a big backlog of PRs. (If you're unsure whether to make a PR, write a detailed issue first.)

  - What if there are two people working on PRs at the same time?
    * Take a look at the issue priority. The PR addressing the higher priority issue should merge first. Make sure you pull the new master after that merge before you push changes for your PR. If both issues are high priority, the one with more time-sensitive commits should be merged first. If you're unsure, ask.
- If we do have a backlog of PRs, it's fine to make a new branch off your current PR, and make a new PR from that. These "cumulative" PRs are not ideal, but they are better than creating merge conflicts with yourself!
- High priority issues are organized in the "Project" view from top (most urgent) to bottom (least urgent) and may be labeled with `highpriority` as appropriate. If you are working on something that is urgent or blocks other development, please set a reasonable deadline for review (can be updated, of course!).
- Keep PRs as small as possible: e.g., one issue, one PR. Small PRs are easier to review and merge.
- Before starting work, always ensure you've pulled from master. If you spend more than a few days on your PR, make sure you pull from master regularly. Before making a PR, ensure that your branch is up to date with master.
- Even if your work isn't ready for a PR, push it regularly. A guiding principle is to commit locally every few minutes and push to your branch every 1-2 hours.
- Do not force-push a branch. If you can't push normally, it usually means you need to pull first.

1.1.4.2 Pull request checklist

1. Make sure tests pass on your PR, and that any new features are covered by tests. If tests don't pass, mark the PR as draft until they do.
2. Make sure you've updated `version.py`.
3. Make sure you've updated the changelog.
4. Do not "squash and merge" pull requests (create a merge commit instead).

1.2 What's new

All notable changes to the codebase are documented in this file. Changes that may result in differences in model output, or are required in order to run an old parameter set with the current version, are flagged with the term "Regression information".

Contents

1.2.1 Version 0.10.10 (2021-11-04)

- Refactored parameters to remove `run_pars` and enable a new R pipeline script.
- *GitHub info*: PR 367

1.2.2 Version 0.10.9 (2021-09-30)

- Added age-based routine immunization to immunity initialization for children under 5.
- *GitHub info*: PR 267

1.2.3 Version 0.10.8 (2021-09-e0)

- Added a `to_pandas()` option in base.py for contacts, infection_log, and save_states analyzer
- *GitHub info*: PR 301

1.2.4 Version 0.10.7 (2021-09-04)

- Refactored the `People` object to use a new filtering-based approach based on FPsim.
- *GitHub info*: PR 266

1.2.5 Version 0.10.6 (2021-08-20)

- Removes duplication between `ps.create_sim()` and `ps.run_sims()`.
- Improves "duty cycle" of `ps.run_sims()` to use all processors.
- *GitHub info*: PR 247

1.2.6 Version 0.10.5 (2021-08-20)

- Updates and simplifies intervention logic.
- *GitHub info*: PR 246

1.2.7 Version 0.10.4 (2021-08-20)

- Adds a `Calibration` class, allowing for `sim.calibrate()`.
- Updates how interventions and analyzers are initialized and run.
- Changes the run logic of the sim to prevent double running, allowing more flexibility with run dates, etc.
- Removes deaths as an output and state since there are none.
- *GitHub info*: PR 244

1.2.8 Version 0.10.3 (2021-08-20)

- Porting improvements/features from Covasim.
- Added git info to all objects (sims, multisims, etc.)
- Added an `options` module, e.g. `ps.options.set(interactive=True)`.
- Allows individual people to be extracted, e.g. `sim.people[12]` will give you an object with all details on the 12th person.
- Ported performance improvements for various methods.
- Allows sim results to be exported directly to pandas dataframes via `sim.to_df()`.
- Allow export of population to networkx via `G = sim.people.to_graph()`.
- Adds flexibility in dealing with population layers.
- *GitHub info*: PR 242

1.2.9 Version 0.10.2 (2021-08-20)

- Fixed bug with WPV being initialized incorrectly.
- Refactoring leading to minor performance improvements (10-20%).
- *GitHub info*: PR 241

1.2.10 Version 0.10.1 (2021-08-20)

- Started changelog.
- *GitHub info*: PR 239

1.3 Code of conduct

1.3.1 Our pledge

We believe that a diverse, equitable, and inclusive environment is essential for producing the best quality software. In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in Poliosim development and the Poliosim community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

1.3.2 Our standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community

- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct that could reasonably be considered inappropriate in a professional setting

1.3.3 Our responsibilities

Poliosim maintainers are responsible for clarifying the standards of acceptable behavior and will take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Poliosim maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

1.3.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing Poliosim or its community. Examples of representing the Poliosim project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

1.3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at Poliosim@idmod.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The Poliosim team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Poliosim maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of Poliosim's leadership.

1.3.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html.

For answers to common questions about this code of conduct, see the FAQ.

1.4 API reference

1.4.1 Subpackages

1.4.1.1 poliosim.data package

Submodules

poliosim.data.country_age_data module

get()
This is the following file:

https://github.com/neherlab/covid19_scenarios/blob/master/src/assets/data/country_age_distribution.json

expressed as a function.

poliosim.data.household_size_data module

get()
This is the following file:

https://population.un.org/household/exceldata/population_division_UN_Houseshold_Size_and_Composition_2019.xlsx

expressed as a function.

poliosim.data.loaders module

Load data

get_country_aliases()
Define aliases for countries with odd names in the data

map_entries(*json*, *location*)
Find a match between the JSON file and the provided location(s).

Parameters

- **json** (*list or dict*) – the data being loaded
- **location** (*list or str*) – the list of locations to pull from

get_age_distribution(*location=None*)
Load age distribution for a given country or countries.

Parameters **location** (*str or list*) – name of the country or countries to load the age distribution for

Returns Numpy array of age distributions, or dict if multiple locations

Return type age_data (array)

get_household_size(*location=None*)
Load household size distribution for a given country or countries.

Parameters **location** (*str or list*) – name of the country or countries to load the age distribution for

Returns Size of household, or dict if multiple locations

Return type house_size (float)

poliosim.data.state_age_data module

`get()`
This data is translated from the US Census csv to the new json format expressed as a function

1.4.2 Submodules

1.4.2.1 poliosim.analysis module

Additional analysis functions.

`class Analyzer(label=None)`
Bases: `sciris.sc_utils.prettyobj`

Base class for analyzers. Based on the Intervention class. Analyzers are used to provide more detailed information about a simulation than is available by default – for example, pulling states out of sim.people on a particular timestep before it gets updated in the next timestep.

To retrieve a particular analyzer from a sim, use sim.get_analyzer().

Parameters `label` (*str*) – a label for the Analyzer (used for ease of identification)

`initialize(sim=None)`
Initialize the analyzer, e.g. convert date strings to integers.

`finalize(sim=None)`
Finalize analyzer

This method is run once as part of *sim.finalize()* enabling the analyzer to perform any final operations after the simulation is complete (e.g. rescaling)

`apply(sim)`
Apply analyzer at each time point. The analyzer has full access to the sim object, and typically stores data/results in itself. This is the core method which each analyzer object needs to implement.

Parameters `sim` – the Sim instance

`shrink(in_place=False)`
Remove any excess stored data from the intervention; for use with sim.shrink().

Parameters `in_place` (*bool*) – whether to shrink the intervention (else shrink a copy)

`to_json()`
Return JSON-compatible representation

Custom classes can't be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. This method will attempt to JSONify each attribute of the intervention, skipping any that fail.

Returns JSON-serializable representation

`class snapshot(days, *args, **kwargs)`
Bases: *poliosim.analysis.Analyzer*

Analyzer that takes a "snapshot" of the sim.people array at specified points in time, and saves them to itself. To retrieve them, you can either access the dictionary directly, or use the get() method.

Parameters

- **days** (*list*) – list of ints/strings/date objects, the days on which to take the snapshot
- **kwargs** (*dict*) – passed to Intervention()

Example:

```
sim = ps.Sim(analyzers=ps.snapshot('2020-04-04', '2020-04-14'))
sim.run()
snapshot = sim['analyzers'][0]
people = snapshot.snapshots[0]             # Option 1
people = snapshot.snapshots['2020-04-04'] # Option 2
people = snapshot.get('2020-04-14')        # Option 3
people = snapshot.get(34)                  # Option 4
people = snapshot.get()                    # Option 5
```

initialize(*sim*)

apply(*sim*)

get(*key=None*)
Retrieve a snapshot from the given key (int, str, or date)

class save_states(*states=None*, *full=False*, *sort_inds=None*, ***kwargs*)
Bases: *poliosim.analysis.Analyzer*

Save the states of the people into a big array for polio.

Parameters

- **states** (*list) the list of states to save (default, just shedding*) –
- **full** (*bool*) – whether to save the complete list of states

initialize(*sim*)

apply(*sim*)

to_dfdict()

plot(*sort_inds=None*, *log_viral_shed=True*, *nonzero_only=False*, *filter_states=None*, *rows=None*, ***kwargs*)

coplot(*people_inds=None*, *filter_states=None*, *transform_state=None*, ***kwargs*)
Co-plot selected states of selected people, one plot per person.

Parameters

- **people_inds** – The indices of the people to create plots for.
- **filter_states** – If specified, only these states will be shown. The states you want also need to be
- **object** (*specified when creating the analyzer*) –
- **didn't** (*or else be one of the defaults if you*) –
- **explicitly.** (*specify any*) –
- **transform_state** – An optional dictionary, keyed on state names. The dictionary value is a function
- **plotted.** (*to pass the state values through. The transformed value will be*) –

- ****kwargs** –

Returns Nothing.

Example of using transform_state:

```
# This function returns log10(x) when x > 0, otherwise np.nan:
def snoop_loggy_log10(x):
    return np.log10(x, where=0 < x, out=np.nan * x)

# Apply the special log function just to the state 'viral_shed'.
my_analyzer.plot(people_inds=inds, transform_state={'viral_shed': snoop_loggy_
↪log10})
```

class track_events(*states=None*, *full=True*, *curr_imm=False*, ***kwargs*)
Bases: *poliosim.analysis.Analyzer*

Store a list of events for everyone.

Parameters

- **states** (*list) the list of states to save (default, just shedding*) –
- **full** (*bool*) – whether to save the complete list of states
- **curr_imm** (*bool*) – whether to store current immunity, which is updated on every timestep (default: false)

Data are stored in self.events for the events, and self.init_state for the initial state.

initialize(*sim*)

finalize(*sim=None*)
Convert from list-of-dicts to dataframe

apply(*sim*)

class track_shedders(***kwargs*)
Bases: *poliosim.analysis.Analyzer*

Keep a record of people who shed

initialize(*sim*)

apply(*sim*)
Record data throughout the sim

plot(*font_family=None*)

class infection_report(*naive_immunity=8.0*, *age_breakpoints=None*, ***kwargs*)
Bases: *poliosim.analysis.Analyzer*

Keep a record of who infects who.

Parameters

- **naive_immunity** (*float*) – immunity threshold for categorization as naive (units = Nab). Default is 8.0.
- **age_breakpoints** (*array of float*) – age breakpoints for making categories.
- **[5** (*Default is*) –
- **u5'** (*'non-naive*) –
- **u5'** –

- **'5-15'** –
- **'o15'.** –
- **variants.** (*All categories with lower bound <5 will have naive and non-naive*) –
- **raised.** (*The first age breakpoint must be >0 or an exception will be*) –

make_infection_report_categories(*age_breakpoints*)

initialize(*sim*)

get_infection_report_category(*age*, *immunity*)

apply(*sim*)
Record data throughout the sim

plot(*font_family=None*)

class Fit(*sim*, *weights=None*, *keys=None*, *method=None*, *custom=None*, *compute=True*, *verbose=False*, ***kwargs*)
Bases: sciris.sc_utils.prettyobj

A class for calculating the fit between the model and the data. Note the following terminology is used here:

- fit: nonspecific term for how well the model matches the data
- difference: the absolute numerical differences between the model and the data (one time series per result)
- goodness-of-fit: the result of passing the difference through a statistical function, such as mean squared error
- loss: the goodness-of-fit for each result multiplied by user-specified weights (one time series per result)
- mismatch: the sum of all the loses (a single scalar value) – this is the value to be minimized during calibration

Parameters

- **sim** (Sim) – the sim object
- **weights** (*dict*) – the relative weight to place on each result
- **keys** (*list*) – the keys to use in the calculation
- **method** (*str*) – the method to be used to calculate the goodness-of-fit
- **custom** (*dict*) – a custom dictionary of additional data to fit; format is e.g. {'<label>':{'data':[1,2,3], 'sim':[1,2,4], 'weights':2.0}}
- **compute** (*bool*) – whether to compute the mismatch immediately
- **verbose** (*bool*) – detail to print
- **kwargs** (*dict*) – passed to compute_gof()

Example:

```
sim = ps.Sim() # Needs data to work
sim.run()
fit = sim.compute_fit()
fit.plot()
```

compute()
Perform all required computations

reconcile_inputs()
Find matching keys and indices between the model and the data

compute_diffs(*absolute=False*)
Find the differences between the sim and the data

compute_gofs(***kwargs*)
Compute the goodness-of-fit

compute_losses()
Compute the weighted goodness-of-fit

compute_mismatch(*use_median=False*)
Compute the final mismatch

plot(*keys=None*, *width=0.8*, *font_size=18*, *fig_args=None*, *axis_args=None*, *plot_args=None*)
Plot the fit of the model to the data. For each result, plot the data and the model; the difference; and the loss (weighted difference). Also plots the loss as a function of time.

Parameters

- **keys** (*list*) – which keys to plot (default, all)
- **width** (*float*) – bar width
- **font_size** (*float*) – size of font
- **fig_args** (*dict*) – passed to pl.figure()
- **axis_args** (*dict*) – passed to pl.subplots_adjust()
- **plot_args** (*dict*) – passed to pl.plot()

class TransTree(*sim*, *to_networkx=False*)
Bases: sciris.sc_utils.prettyobj

A class for holding a transmission tree. There are several different representations of the transmission tree: "infection_log" is copied from the people object and is the simplest representation. "detailed h" includes additional attributes about the source and target. If NetworkX is installed (required for most methods), "graph" includes an NX representation of the transmission tree.

Parameters

- **sim** (Sim) – the sim object
- **to_networkx** (*bool*) – whether to convert the graph to a NetworkX object

property transmissions
Iterable over edges corresponding to transmission events

This excludes edges corresponding to seeded infections without a source

day(*day=None*, *which=None*)
Convenience function for converting an input to an integer day

count_targets(*start_day=None*, *end_day=None*)
Count the number of targets each infected person has. If start and/or end days are given, it will only count the targets of people who got infected between those dates (it does not, however, filter on the date the target got infected).

Parameters

- **start_day** (*int/str*) – the day on which to start counting people who got infected
- **end_day** (*int/str*) – the day on which to stop counting people who got infected

make_detailed(*people*, *reset=False*)
Construct a detailed transmission tree, with additional information for each person

plot(**args*, ***kwargs*)
Plot the transmission tree

plot_histograms(*start_day=None*, *end_day=None*, *bins=None*, *width=0.8*, *fig_args=None*, *font_size=18*)
Plots a histogram of the number of transmissions.

Parameters

- **start_day** (*int/str*) – the day on which to start counting people who got infected
- **end_day** (*int/str*) – the day on which to stop counting people who got infected
- **bins** (*list*) – bin edges to use for the histogram
- **width** (*float*) – width of bars
- **fig_args** (*dict*) – passed to pl.figure()
- **font_size** (*float*) – size of font

plot_intervals()
Plot serial interval distribution

class calculate_contacts_infected(*ignore_last_n_days=None*, ***kwargs*)
Bases: *poliosim.analysis.Analyzer*

Calculate the contacts infected

apply(*sim*)

calculate_contact_counts(*sim*)

calculate_contacts_infected(*sim*)

clear()

plot()

save(*folder=None*)

1.4.2.2 poliosim.base module

Base classes for Poliosim. These classes handle a lot of the boilerplate of the People and Sim classes (e.g. loading, saving, key lookups, etc.), so those classes can be focused on the disease-specific functionality.

class ParsObj(*pars*)
Bases: sciris.sc_utils.prettyobj

A class based around performing operations on a self.pars dict.

update_pars(*pars=None*, *create=False*)
Update internal dict with new pars.

Parameters

- **pars** (*dict*) – the parameters to update (if None, do nothing)
- **create** (*bool*) – if create is False, then raise a KeyNotFoundError if the key does not already exist

class Result(*name=None*, *npts=None*, *scale='dynamic'*, *color=None*)
Bases: `object`

Stores a single result – by default, acts like an array.

Parameters

- **name** (*str*) – name of this result, e.g. new_infections
- **npts** (*int*) – if values is None, precreate it to be of this length
- **scale** (*str*) – whether or not the value scales by population size; options are "dynamic", "static", or False
- **color** (*str/arr*) – default color for plotting (hex or RGB notation)
- **n_variants** (*int*) – the number of variants the result is for (0 for results not by variant)

Example:

```
import poliosim as ps
r1 = ps.Result(name='test1', npts=10)
r1[:5] = 20
print(r1.values)
```

property npts

class BaseSim(**args*, ***kwargs*)
Bases: *poliosim.base.ParsObj*

The BaseSim class handles the running of the simulation: the number of people, number of time points, and the parameters of the simulation.

set_metadata(*simfile*)
Set the metadata for the simulation – creation time and filename

set_seed(*seed=- 1*)
Set the seed for the random number stream from the stored or supplied value

Parameters **seed** (*None or int*) – if no argument, use current seed; if None, randomize; otherwise, use and store supplied seed

Returns None

property n
Count the number of people – if it fails, assume none

property scaled_pop_size
Get the total population size, i.e. the number of agents times the scale factor – if it fails, assume none

property npts
Count the number of time points

property tvec
Create a time vector

property datevec
Create a vector of dates

Returns Array of *datetime* instances containing the date associated with each simulation time step

day(*day*, **args*)
Convert a string, date/datetime object, or int to a day (int).

Parameters **day** (*str, date, int, or list*) – convert any of these objects to a day relative to the simulation's start day

Returns the day(s) in simulation time

Return type days (int or str)

Example:

```
sim.day('2020-04-05') # Returns 35
```

date(*ind*, **args*, *dateformat=None*, *as_date=False*)

Convert one or more integer days of simulation time to a date/list of dates – by default returns a string, or returns a datetime Date object if as_date is True. See also ps.date(), which provides a partly overlapping set of date conversion features.

Parameters

- **ind** (*int, list, or array*) – the day(s) in simulation time
- **args** (*list*) – additional day(s)
- **dateformat** (*str*) – the format to return the date in
- **as_date** (*bool*) – whether to return as a datetime date instead of a string

Returns the date(s) corresponding to the simulation day(s)

Return type dates (str, Date, or list)

Examples:

```
sim.date(34) # Returns '2020-04-04'
sim.date([34, 54]) # Returns ['2020-04-04', '2020-04-24']
sim.date(34, 54, as_dt=True) # Returns [datetime.date(2020, 4, 4), datetime.
↪date(2020, 4, 24)]
```

result_keys()

Get the actual results objects, not other things stored in sim.results

copy()

Returns a deep copy of the sim

export_results(*for_json=True*, *filename=None*, *indent=2*, **args*, ***kwargs*)

Convert results to dict – see also to_json().

The results written to Excel must have a regular table shape, whereas for the JSON output, arbitrary data shapes are supported.

Parameters

- **for_json** (*bool*) – if False, only data associated with Result objects will be included in the converted output
- **filename** (*str*) – filename to save to; if None, do not save
- **indent** (*int*) – indent (int): if writing to file, how many indents to use per nested level
- **args** (*list*) – passed to savejson()
- **kwargs** (*dict*) – passed to savejson()

Returns dictionary representation of the results

Return type resdict (dict)

export_pars(*filename=None*, *indent=2*, **args*, ***kwargs*)
Return parameters for JSON export – see also to_json().

This method is required so that interventions can specify their JSON-friendly representation.

Parameters

- **filename** (*str*) – filename to save to; if None, do not save
- **indent** (*int*) – indent (int): if writing to file, how many indents to use per nested level
- **args** (*list*) – passed to savejson()
- **kwargs** (*dict*) – passed to savejson()

Returns a dictionary containing all the parameter values

Return type pardict (dict)

to_json(*filename=None*, *keys=None*, *tostring=False*, *indent=2*, *verbose=False*, **args*, ***kwargs*)
Export results and parameters as JSON.

Parameters

- **filename** (*str*) – if None, return string; else, write to file
- **keys** (*str or list*) – attributes to write to json (default: results, parameters, and summary)
- **tostring** (*bool*) – if not writing to file, whether to write to string (alternative is sanitized dictionary)
- **indent** (*int*) – if writing to file, how many indents to use per nested level
- **verbose** (*bool*) – detail to print
- **args** (*list*) – passed to savejson()
- **kwargs** (*dict*) – passed to savejson()

Returns A unicode string containing a JSON representation of the results, or writes the JSON file to disk

Examples:

```
json = sim.to_json()
sim.to_json('results.json')
sim.to_json('summary.json', keys='summary')
```

to_df(*date_index=False*)
Export results to a pandas dataframe

Parameters **date_index** (*bool*) – if True, use the date as the index

to_pandas(*key=None*, *to_file=False*, *filename='output.ftr'*)
Return a part of the sim as a pandas dataframe with optional export

Parameters

- **key** (*str*) – attribute to return as pandas dataframe (options are: contacts, infection_log, or states (from save_states analyzer))
- **to_file** (*bool*) – if True, export as a feather file
- **filename** (*str*) – filename for write to file (should end in .ftr)

Returns A pd.dataframe, or writes the file to disk

Examples:

```
sim = ps.create_sim(rand_seed=100, pop_size=10e3, n_days=30,
            pop_infected = 100,
            vx_coverage=0.05,
            trace_prob=1.0, test_delay=0,
            rel_symp_prob=1.0,
            analyzers=ps.save_states(full=True),
        )
sim.run()
sim.to_pandas('contacts')
sim.to_pandas('contacts', to_file=True, filename='df_contacts.ftr')
sim.to_pandas('infection_log', to_file=True, filename='df_infection_log.ftr')
sim.to_pandas('states', to_file=True, filename='df_states.ftr')
```

to_excel(*filename=None*, *skip_pars=None*)
Export parameters and results as Excel format

Parameters

- **filename** (*str*) – if None, return string; else, write to file
- **skip_pars** (*list*) – if provided, a custom list parameters to exclude

Returns An sc.Spreadsheet with an Excel file, or writes the file to disk

shrink(*skip_attrs=None*, *in_place=True*)
"Shrinks" the simulation by removing the people, and returns a copy of the "shrunken" simulation. Used to reduce the memory required for saved files.

Parameters **skip_attrs** (*list*) – a list of attributes to skip in order to perform the shrinking; default "people"

Returns a Sim object with the listed attributes removed

Return type shrunken (*Sim*)

save(*filename=None*, *keep_people=None*, *skip_attrs=None*, ***kwargs*)
Save to disk as a gzipped pickle.

Parameters

- **filename** (*str or None*) – the name or path of the file to save to; if None, uses stored
- **kwargs** – passed to sc.makefilepath()

Returns the validated absolute path to the saved file

Return type filename (str)

Example:

```
sim.save() # Saves to a .sim file with the date and time of creation by default
```

static load(*filename*, **args*, ***kwargs*)
Load from disk from a gzipped pickle.

Parameters

- **filename** (*str*) – the name or path of the file to load from
- **kwargs** – passed to ps.load()

Returns the loaded simulation object

Return type sim (*Sim*)

Example:

```
sim = ps.Sim.load('my-simulation.sim')
```

update_pars(*pars=None*, *create=False*, ***kwargs*)
Ensure that metaparameters get used properly before being updated

load_data(*datafile=None*, *datacols=None*, *verbose=None*, ***kwargs*)
Load the data to calibrate against, if provided

layer_keys()
Attempt to retrieve the current layer keys, in the following order: from the people object (for an initialized sim), from the popdict (for one in the process of being initialized), from the beta_layer parameter (for an uninitialized sim), or by assuming a default (if none of the above are available).

validate_layer_pars()
Handle layer parameters, since they need to be validated after the population creation, rather than before.

validate_pars(*validate_layers=True*)
Some parameters can take multiple types; this makes them consistent.

Parameters **validate_layers** (*bool*) – whether to validate layer parameters as well via validate_layer_pars() – usually yes, except during initialization

load_population(*popfile=None*, ***kwargs*)
Load the population dictionary from file – typically done automatically as part of sim.initialize(). Supports loading either saved population dictionaries (popdicts, file ending .pop by convention), or ready-to-go People objects (file ending .ppl by convention). Either object an also be supplied directly. Once a population file is loaded, it is removed from the Sim object.

Parameters

- **popfile** (*str or obj*) – if a string, name of the file; otherwise, the popdict or People object to load
- **kwargs** (*dict*) – passed to sc.makefilepath()

init_people(*save_pop=False*, *load_pop=False*, *popfile=None*, *verbose=None*, *seed_infections=None*, ***kwargs*)
Create the people.

Parameters

- **save_pop** (*bool*) – if true, save the population dictionary to popfile
- **load_pop** (*bool*) – if true, load the population dictionary from popfile
- **popfile** (*str*) – filename to load/save the population
- **verbose** (*int*) – detail to print
- **kwargs** (*dict*) – passed to ps.make_people()

init_interventions()
Initialize and validate the interventions

finalize_interventions()

init_analyzers()
Initialize the analyzers

finalize_analyzers()

restore_pars(*orig_pars*)
Restore the original parameter values, except for the analyzers

compute_summary(*verbose=None*)
Compute the summary statistics to display at the end of a run

summarize(*output=False*)
Print a brief summary of the simulation

brief(*output=False*)
Return a one-line description of a sim

compute_fit(**args*, ***kwargs*)
Compute the fit between the model and the data. See cv.Fit() for more information.

Parameters

- **args** (*list*) – passed to cv.Fit()
- **kwargs** (*dict*) – passed to cv.Fit()

Returns A Fit object

Example:

```
sim = cv.Sim(datafile='data.csv')
sim.run()
fit = sim.compute_fit()
fit.plot()
```

calibrate(*calib_pars*, ***kwargs*)
Automatically calibrate the simulation, returning a Calibration object (a type of analyzer). See the documentation on that class for more information.

Parameters

- **calib_pars** (*dict*) – a dictionary of the parameters to calibrate of the format dict(key1=[best, low, high])
- **kwargs** (*dict*) – passed to cv.Calibration()

Returns A Calibration object

Example:

```
sim = cv.Sim(datafile='data.csv')
calib_pars = dict(beta=[0.015, 0.010, 0.020])
calib = sim.calibrate(calib_pars, n_trials=50)
calib.plot()
```

make_age_histogram(*output=True*, **args*, ***kwargs*)
Calculate the age histograms of infections, diagnoses, etc. See ps.age_histogram() for more information. This can be used alternatively to supplying the age histogram as an analyzer to the sim. If used this way, it can only record the final time point since the states of each person are not saved during the sim.

Parameters

- **output** (*bool*) – whether or not to return the age histogram; if not, store in sim.results
- **args** (*list*) – passed to ps.age_histogram()
- **kwargs** (*dict*) – passed to ps.age_histogram()

Example:

```
sim = ps.Sim()
sim.run()
agehist = sim.make_age_histogram()
agehist.plot()
```

make_transtree(*output=True*, **args*, ***kwargs*)

Create a TransTree (transmission tree) object, for analyzing the pattern of transmissions in the simulation. See ps.TransTree() for more information.

Parameters

- **output** (*bool*) – whether or not to return the TransTree; if not, store in sim.results
- **args** (*list*) – passed to ps.TransTree()
- **kwargs** (*dict*) – passed to ps.TransTree()

Example:

```
sim = ps.Sim()
sim.run()
tt = sim.make_transtree()
```

plot(**args*, ***kwargs*)

Plot the results of a single simulation.

plot_result(*key*, **args*, ***kwargs*)

Simple method to plot a single result. Useful for results that aren't standard outputs. See sim.plot() for explanation of other arguments.

Parameters **key** (*str*) – the key of the result to plot

Examples:

```
sim.plot_result('r_eff')
```

get_interventions(*label=None*, *partial=False*, *as_inds=False*)

Find the matching intervention(s) by label, index, or type. If None, return all interventions. If the label provided is "summary", then print a summary of the interventions (index, label, type).

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. 'beta' will match all beta interventions)
- **as_inds** (*bool*) – if true, return matching indices instead of the actual interventions

Examples:

```
tp = ps.test_prob(symp_prob=0.1)
cb1 = ps.change_beta(days=5, changes=0.3, label='NPI')
cb2 = ps.change_beta(days=10, changes=0.3, label='Masks')
sim = ps.Sim(interventions=[tp, cb1, cb2])
cb1, cb2 = sim.get_interventions(ps.change_beta)
tp, cb2 = sim.get_interventions([0,2])
ind = sim.get_interventions(ps.change_beta, as_inds=True) # Returns [1,2]
sim.get_interventions('summary') # Prints a summary
```

get_intervention(*label=None*, *partial=False*, *first=False*, *die=True*)
Like get_interventions(), find the matching intervention(s) by label, index, or type. If more than one intervention matches, return the last by default. If no label is provided, return the last intervention in the list.

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. 'beta' will match all beta interventions)
- **first** (*bool*) – if true, return first matching intervention (otherwise, return last)
- **die** (*bool*) – whether to raise an exception if no intervention is found

Examples:

```
tp = ps.test_prob(symp_prob=0.1)
cb = ps.change_beta(days=5, changes=0.3, label='NPI')
sim = ps.Sim(interventions=[tp, cb])
cb = sim.get_intervention('NPI')
cb = sim.get_intervention('NP', partial=True)
cb = sim.get_intervention(ps.change_beta)
cb = sim.get_intervention(1)
cb = sim.get_intervention()
tp = sim.get_intervention(first=True)
```

get_analyzers(*label=None*, *partial=False*, *as_inds=False*)
Same as get_interventions(), but for analyzers.

get_analyzer(*label=None*, *partial=False*, *first=False*, *die=True*)
Same as get_intervention(), but for analyzers.

class BasePeople
Bases: sciris.sc_utils.prettyobj

A class to handle all the boilerplate for people – note that everything interesting happens in the People class.

Parameters **pars** (*dict*) – a dictionary with, at minimum, keys 'pop_size' and 'n_days'

Initialize essential attributes used for filtering

property len_inds
Alias (almost) to len(self)

property len_people
Full length of People array, ignoring filtering

property pop_size
Alias to len_people

property inds
Alias to self._inds to prevent accidental overwrite & increase speed by allowing "_" shortcircuit

property subinds
Alias to self._subinds

set(*key*, *value*, *die=True*)
Ensure sizes and dtypes match

get(*key*)
Convenience method – key can be string or list of strings

filter(*criteria=None*, *inds=None*, *reset=False*)
Store indices to allow for easy filtering of the People object.

Parameters

- **criteria** (*array*) – a boolean array for the filtering critria
- **inds** (*array*) – alternatively, explicitly filter by these indices
- **reset** (*bool*) – reset the indices rather than use existing ones

Returns A filtered People object, which works just like a normal People object except only operates on a subset of indices.

unfilter()
An easy way of unfiltering the People object, returning the original.

binomial(*prob*, *as_inds=False*, *as_filter=True*)
Return indices either by a single probability or by an array of probabilities. By default just return the boolean array, but can also return the indices, or the filtered People object.

Parameters

- **prob** (*float/array*) – either a scalar probability, or an array of probabilities of the same length as People
- **as_inds** (*bool*) – return as list of indices instead of a boolean array
- **as_filter** (*bool*) – return as filter instead than boolean array

filter_tf(*arr*)
Split the People object into True and False sets

true(*key*)
Return indices matching the condition

false(*key*)
Return indices not matching the condition

defined(*key*)
Return indices of people who are not-nan

not_defined(*key*)
Return indices of people who are nan

count(*key*)
Count the number of people for a given key

count_not(*key*)
Count the number of people who do not have a property for a given key

set_pars(*pars=None*)
Re-link the parameters stored in the people object to the sim containing it, and perform some basic validation.

keys()
Returns keys for all properties of the people object

person_keys()
Returns keys specific to a person (e.g., their age)

state_keys()
Returns keys for different states of a person (e.g., symptomatic)

date_keys()
Returns keys for different event dates (e.g., date a person became symptomatic)

dur_keys()
Returns keys for different durations (e.g., the duration from exposed to infectious)

layer_keys()
Get the available contact keys – try contacts first, then beta_layer

indices()
The indices of each people array

validate(*die=True*, *verbose=False*)

to_df()
Convert to a Pandas dataframe

to_arr()
Return as numpy array

person(*ind*)
Method to create person from the people

to_people()
Return all people as a list

from_people(*people*, *resize=True*)
Convert a list of people back into a People object

to_graph()
Convert all people to a networkx MultiDiGraph, including all properties of the people (nodes) and contacts (edges).

Example:

```
import poliosim as ps
import networkx as nx
sim = ps.Sim(pop_size=50, pop_type='hybrid', contacts=dict(h=3, s=10, w=10,␣
↪c=5)).run()
G = sim.people.to_graph()
nodes = G.nodes(data=True)
edges = G.edges(keys=True)
node_colors = [n['age'] for i,n in nodes]
layer_map = dict(h='#37b', s='#e11', w='#4a4', c='#a49')
edge_colors = [layer_map[G[i][j][k]['layer']] for i,j,k in edges]
edge_weights = [G[i][j][k]['beta']*5 for i,j,k in edges]
nx.draw(G, node_color=node_colors, edge_color=edge_colors, width=edge_weights,␣
↪alpha=0.5)
```

init_contacts(*reset=False*)
Initialize the contacts dataframe with the correct columns and data types

add_contacts(*contacts*, *lkey=None*, *beta=None*)
Add new contacts to the array

update_contacts()
Refresh dynamic contacts, e.g. community

make_edgelist(*contacts*)
Parse a list of people with a list of contacts per person and turn it into an edge list.

static remove_duplicates(*df*)
Sort the dataframe and remove duplicates – note, not extensively tested

class Person(*pars=None*, *uid=None*, *age=- 1*, *sex=- 1*, *contacts=None*)
Bases: `sciris.sc_utils.prettyobj`

Class for a single person. Note: this is largely deprecated since sim.people is now based on arrays rather than being a list of people.

class FlexDict
Bases: `dict`

A dict that allows more flexible element access: in addition to obj['a'], also allow obj[0]. Lightweight implementation of the Sciris odict class.

keys()

values()

items()

class Contacts(*layer_keys=None*)
Bases: *poliosim.base.FlexDict*

A simple (for now) class for storing different contact layers.

add_layer(***kwargs*)
Small method to add one or more layers to the contacts. Layers should be provided as keyword arguments.

Example:

```
hospitals_layer = cv.Layer(label='hosp')
sim.people.contacts.add_layer(hospitals=hospitals_layer)
```

pop_layer(**args*)
Remove the layer(s) from the contacts.

Example:

```
sim.people.contacts.pop_layer('hospitals')
```

Note: while included here for convenience, this operation is equivalent to simply popping the key from the contacts dictionary.

to_graph()
Convert all layers to a networkx MultiDiGraph

Example:

```
import networkx as nx
sim = cv.Sim(pop_size=50, pop_type='hybrid').run()
G = sim.people.contacts.to_graph()
nx.draw(G)
```

class Layer(*label=None*, ***kwargs*)
Bases: *poliosim.base.FlexDict*

A small class holding a single layer of contact edges (connections) between people.

The input is typically three arrays: person 1 of the connection, person 2 of the connection, and the weight of the connection. Connections are undirected; each person is both a source and sink.

This class is usually not invoked directly by the user, but instead is called as part of the population creation.

Parameters

- **p1** (*array*) – an array of N connections, representing people on one side of the connection
- **p2** (*array*) – an array of people on the other side of the connection
- **beta** (*array*) – an array of weights for each connection
- **label** (*str*) – the name of the layer (optional)
- **kwargs** (*dict*) – other keys copied directly into the layer

Note that all arguments (except for label) must be arrays of the same length, although not all have to be supplied at the time of creation (they must all be the same at the time of initialization, though, or else validation will fail).

Examples:

```
# Generate an average of 10 contacts for 1000 people
n = 10_000
n_people = 1000
p1 = np.random.randint(n_people, size=n)
p2 = np.random.randint(n_people, size=n)
beta = np.ones(n)
layer = cv.Layer(p1=p1, p2=p2, beta=beta, label='rand')

# Convert one layer to another with extra columns
index = np.arange(n)
self_conn = p1 == p2
layer2 = cv.Layer(**layer, index=index, self_conn=self_conn, label=layer.label)
```

property members
Return sorted array of all members

meta_keys()
Return the keys for the layer's meta information – i.e., p1, p2, beta

validate()
Check the integrity of the layer: right types, right lengths

pop_inds(*inds*)
"Pop" the specified indices from the edgelist and return them as a dict. Returns in the right format to be used with layer.append().

Parameters **inds** (*int, array, slice*) – the indices to be removed

append(*contacts*)
Append contacts to the current layer.

Parameters **contacts** (*dict*) – a dictionary of arrays with keys p1,p2,beta, as returned from layer.pop_inds()

to_df()
Convert to dataframe

from_df(*df*, *keys=None*)
Convert from a dataframe

to_graph()
Convert to a networkx DiGraph

Example:

```
import networkx as nx
sim = ps.Sim(pop_size=20, pop_type='hybrid').run()
G = sim.people.contacts['h'].to_graph()
nx.draw(G)
```

find_contacts(*inds*, *as_array=True*)
Find all contacts of the specified people

For some purposes (e.g. contact tracing) it's necessary to find all of the contacts associated with a subset of the people in this layer. Since contacts are bidirectional it's necessary to check both P1 and P2 for the target indices. The return type is a Set so that there is no duplication of indices (otherwise if the Layer has explicit symmetric interactions, they could appear multiple times). This is also for performance so that the calling code doesn't need to perform its own unique() operation. Note that this cannot be used for cases where multiple connections count differently than a single infection, e.g. exposure risk.

Parameters

- **inds** (*array*) – indices of people whose contacts to return
- **as_array** (*bool*) – if true, return as sorted array (otherwise, return as unsorted set)

Returns a set of indices for pairing partners

Return type contact_inds (array)

Example: If there were a layer with - P1 = [1,2,3,4] - P2 = [2,3,1,4] Then find_contacts([1,3]) would return {1,2,3}

update(*people*, *frac=1.0*)
Regenerate contacts on each timestep.

This method gets called if the layer appears in `sim.pars['dynam_layer']`. The Layer implements the update procedure so that derived classes can customize the update e.g. implementing over-dispersion/other distributions, random clusters, etc.

Typically, this method also takes in the `people` object so that the update can depend on person attributes that may change over time (e.g. changing contacts for people that are severe/critical).

Parameters

- **people** (People) – the Poliosim People object, which is usually used to make new contacts
- **frac** (*float*) – the fraction of contacts to update on each timestep

1.4.2.3 poliosim.interventions module

Specify the core interventions available in Poliosim. Other interventions can be defined by the user by inheriting from these classes.

InterventionDict(*which*, *pars*)
Generate an intervention from a dictionary. Although a function, it acts like a class, since it returns a class instance.

Example:

```
interv = ps.InterventionDict(which='change_beta', pars={'days': 30, 'changes': 0.5,
↪'layers': None})
```

(continues on next page)

(continued from previous page)

class Intervention(*label=None*, *show_label=False*, *do_plot=None*, *line_args=None*)
Bases: object

Base class for interventions. By default, interventions are printed using a dict format, which they can be recreated from. To display all the attributes of the intervention, use disp() instead.

To retrieve a particular intervention from a sim, use sim.get_intervention().

Parameters

- **label** (*str*) – a label for the intervention (used for plotting, and for ease of identification)
- **show_label** (*bool*) – whether or not to include the label in the legend
- **do_plot** (*bool*) – whether or not to plot the intervention
- **line_args** (*dict*) – arguments passed to pl.axvline() when plotting

disp()
Print a detailed representation of the intervention

initialize(*sim=None*)
Initialize intervention – this is used to make modifications to the intervention that can't be done until after the sim is created.

finalize(*sim=None*)
Finalize intervention

This method is run once as part of *sim.finalize()* enabling the intervention to perform any final operations after the simulation is complete (e.g. rescaling)

apply(*sim*)
Apply the intervention. This is the core method which each derived intervention class must implement. This method gets called at each timestep and can make arbitrary changes to the Sim object, as well as storing or modifying the state of the intervention.

Parameters **sim** – the Sim instance

Returns None

shrink(*in_place=False*)
Remove any excess stored data from the intervention; for use with sim.shrink().

Parameters **in_place** (*bool*) – whether to shrink the intervention (else shrink a copy)

plot_intervention(*sim*, *ax=None*, ***kwargs*)
Plot the intervention

This can be used to do things like add vertical lines on days when interventions take place. Can be disabled by setting self.do_plot=False.

Note 1: you can modify the plotting style via the `line_args` argument when creating the intervention.

Note 2: By default, the intervention is plotted at the days stored in self.days. However, if there is a self.plot_days attribute, this will be used instead.

Parameters

- **sim** – the Sim instance
- **ax** – the axis instance
- **kwargs** – passed to ax.axvline()

Returns None

`to_json()`
Return JSON-compatible representation

Custom classes can't be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. In the first instance, the object dict will be returned. However, if an intervention itself contains non-standard variables as attributes, then its *to_json* method will need to handle those.

Note that simply printing an intervention will usually return a representation that can be used to recreate it.

Returns JSON-serializable representation (typically a dict, but could be anything else)

`class change_beta(days, changes, layers=None, **kwargs)`
Bases: *poliosim.interventions.Intervention*

The most basic intervention – change beta by a certain amount.

Parameters

- **days** (*int or array*) – the day or array of days to apply the interventions
- **changes** (*float or array*) – the changes in beta (1 = no change, 0 = no transmission)
- **layers** (*str or list*) – the layers in which to change beta
- **kwargs** (*dict*) – passed to Intervention()

Examples:

```
interv = ps.change_beta(25, 0.3) # On day 25, reduce overall beta by 70% to 0.3
interv = ps.change_beta([14, 28], [0.7, 1], layers='s') # On day 14, reduce beta by␣
↪30%, and on day 28, return to 1 for schools
```

`initialize(sim)`
Fix days and store beta

`apply(sim)`

`class test_prob(symp_prob, asymp_prob=0.0, symp_quar_prob=None, asymp_quar_prob=None, quar_policy=None, subtarget=None, test_sensitivity=1.0, loss_prob=0.0, test_delay=0, start_day=0, end_day=None, **kwargs)`
Bases: *poliosim.interventions.Intervention*

Test as many people as required based on test probability. Probabilities are OR together, so choose wisely.

Parameters

- **symp_prob** (*float*) – Probability of testing a symptomatic (unquarantined) person
- **asymp_prob** (*float*) – Probability of testing an asymptomatic (unquarantined) person
- **symp_quar_prob** (*float*) – Probability of testing a symptomatic quarantined person
- **asymp_quar_prob** (*float*) – Probability of testing an asymptomatic quarantined person
- **quar_policy** (*str*) – Policy for testing in quarantine: options are 'start', 'end', 'both' (start and end), 'daily'
- **subtarget** (*dict*) – subtarget intervention to people with particular indices (see test_num() for details)
- **test_sensitivity** (*float*) – Probability of a true positive
- **loss_prob** (*float*) – Probability of loss to follow-up

- **test_delay** (*int*) – How long testing takes
- **start_day** (*int*) – When to start the intervention
- **kwargs** (*dict*) – passed to Intervention()

Examples:

```
interv = ps.test_prob(symp_prob=0.1, asymp_prob=0.01) # Test 10% of symptomatics␣
↪and 1% of asymptomatics
interv = ps.test_prob(symp_quar_prob=0.4) # Test 40% of those in quarantine with␣
↪symptoms
```

initialize(*sim*)
Fix the dates

apply(*sim*)
Perform testing

class contact_tracing(*trace_probs=None*, *trace_time=None*, *start_day=0*, *end_day=None*, *presumptive=False*, ***kwargs*)
Bases: *poliosim.interventions.Intervention*

Contact tracing of positive people.

Parameters

- **trace_probs** (*dict*) – probability of tracing, per layer
- **trace_time** (*dict*) – days required to trace, per layer
- **start_day** (*int*) – intervention start day
- **end_day** (*int*) – intervention end day
- **test_delay** (*int*) – number of days a test result takes
- **presumptive** (*bool*) – whether or not to begin isolation and contact tracing on the presumption of a positive diagnosis
- **kwargs** (*dict*) – passed to Intervention()

initialize(*sim*)
Fix the dates and dictionaries

apply(*sim*)

class symptomatic_triggered_surveillance(*num_sample_mu=10.0*, *num_sample_sigma=0.0*, *test_sensitivity=1.0*, *test_delay=0*, ***kwargs*)
Bases: *poliosim.interventions.test_prob*

Trigger surveillance whenever one or more newly symptomatic cases occur in a timestep. Surveillance is implemented as randomly choosing a set of individuals in the population, as configured by parameter. The other parameters associated with test_priob are also available.

apply(*sim*)

1.4.2.4 poliosim.model module

Defines the full Poliosim model, which consists of the states and functions for updating states of each person (People class), and the dynamics of the simulation (Sim class).

class States

Bases: sciris.sc_utils.prettyobj

For storing all the keys relating to a person and people

person = ['uid', 'age', 'sex', 'symp_prob', 'viral_shed', 'household_id', 'strain_type', 't_infection', 'current_immunity', 'prechallenge_immunity', 'postchallenge_peak_immunity', 'shed_duration', 'log10_peak_cid50', 'exposure_count', 'sus_para_exp_count', 'ri_dose_count', 'ri_take_count', 'sia_dose_count', 'sia_take_count', 'rel_trans']

states = ['naive', 'exposed', 'symptomatic', 'susceptible_to_paralysis', 'IPV_naive', 'is_shed', 'tested', 'diagnosed', 'recovered', 'known_contact', 'quarantined']

dates = ['date_naive', 'date_exposed', 'date_symptomatic', 'date_susceptible_to_paralysis', 'date_IPV_naive', 'date_is_shed', 'date_tested', 'date_diagnosed', 'date_recovered', 'date_known_contact', 'date_quarantined', 'date_pos_test', 'date_end_quarantine', 'date_first_exposed', 'date_end_isolation']

durs = ['dur_exp2inf', 'dur_inf2sym', 'dur_disease']

all_states = ['uid', 'age', 'sex', 'symp_prob', 'viral_shed', 'household_id', 'strain_type', 't_infection', 'current_immunity', 'prechallenge_immunity', 'postchallenge_peak_immunity', 'shed_duration', 'log10_peak_cid50', 'exposure_count', 'sus_para_exp_count', 'ri_dose_count', 'ri_take_count', 'sia_dose_count', 'sia_take_count', 'rel_trans', 'naive', 'exposed', 'symptomatic', 'susceptible_to_paralysis', 'IPV_naive', 'is_shed', 'tested', 'diagnosed', 'recovered', 'known_contact', 'quarantined', 'date_naive', 'date_exposed', 'date_symptomatic', 'date_susceptible_to_paralysis', 'date_IPV_naive', 'date_is_shed', 'date_tested', 'date_diagnosed', 'date_recovered', 'date_known_contact', 'date_quarantined', 'date_pos_test', 'date_end_quarantine', 'date_first_exposed', 'date_end_isolation', 'dur_exp2inf', 'dur_inf2sym', 'dur_disease']

class People(*pars=None*, *strict=True*, ***kwargs*)

Bases: *poliosim.base.BasePeople*

A class to perform all the operations on the people. This class is usually not invoked directly, but instead is created automatically by the sim. Most initialization happens in BasePeople.

Parameters

- **pars** (*dict*) – the sim parameters, e.g. sim.pars – must have pop_size and n_days keys
- **strict** (*bool*) – whether or not to only create keys that are already in self.meta.person; otherwise, let any key be set
- **kwargs** (*dict*) – the actual data, e.g. from a popdict, being specified

initialize(*seed_infections=None*)

Perform initializations

seed_infections(*target_inds=None*)

Create the seed infections

set_prognoses()
Set the prognoses for each person based on age during initialization. Need to reset the seed because viral loads are drawn stochastically.

update_states_pre(*t*)
Perform all state updates at the current timestep

update_states_post()
Perform post-timestep updates

get_sabin_betas()

attempt_canonical_infection(*attempt_date*, *dose=1000000*, *beta=1.0*)
Determine whether candidate targets will be infected based on their current immunity state, and if so, infect them. Uses unmodified betas and infection probabilities from the reference model.

reset_polio()
This isn't used internally, but is used by user scripts

update_polio(*t_days=None*, *inds=None*)
Update each polio agent

check_symptomatic()
Check for new progressions to symptomatic

check_recovered()
Check for and/or do recovery = not infectious any more

check_diagnosed()
Check for new diagnoses. Since most data are reported with diagnoses on the date of the test, this function reports counts not for the number of people who received a positive test result on a day, but rather, the number of people who were tested on that day who are schedule to be diagnosed in the future.

get_quarantine_subtargets(*quarantine_subtarget*, *quarantine_candidate_inds*)
A small helper function.

Parameters

- **quarantine_subtarget** – a function. This function should take (people, quarantine_candidate_inds),
- **'quarantine_candidate_inds'.** (*and should return a dictionary with keys 'inds' and 'vals'. The key 'inds' should be a copy of*) –
- **candidates.** (*The 'vals' should be the associated probability of quarantining each of the quarantine*) –
- **quarantine_candidate_inds** – array of the indices of the candidates to potentially be quarantined.

check_quar()
Check for who gets put into quarantine

make_naive()
Make person naive. This is used during dynamic resampling

polio_infect(*inds*)
State changes associated with polio infection. Indices rather than filtering are used here for speed.

infect(*inds*, *source=None*, *layer=None*)
Infect people and determine their eventual outcomes. Polio-specific state changes are in polio_infect(). Indices rather than filtering are used here for speed.

Parameters

- **inds** (*array*) – array of people to infect
- **source** (*array*) – source indices of the people who transmitted this infection (None if an importation or seed infection)
- **layer** (*str*) – contact layer this infection was transmitted on

Returns number of people infected

Return type count (int)

calculate_peak_immunity(*inds*, *a=4.82*, *b=- 0.3*, *c=3.31*, *d=- 0.32*)
immunity immediately post infection

calculate_current_immunity(*inds*, *rate=0.87*)
immunity after t months have passed since exposure

calculate_shed_duration(*inds*, *u=30.3*, *delta=1.16*, *sigma=1.86*)
probability of shedding given Nab at time t (days post infection); assumes that individual was infected at t = 0; time is measured in days Equation S1 in Famulare 2018 PLOS Bio paper delta_t = time (days) since last infection – survival curve follows lognormal distribution

calculate_viral_shed(*eta=1.65*, *v=0.17*, *epsilon=0.32*)
virus shed per gram, time is in months!

calculate_log10_peak_cid50(*k=0.056*, *Smax=6.7*, *Smin=4.3*, *tau=12*)
returns the peak log10(cid50/g) given prior immunity, age is in months!

test(*inds*, *test_sensitivity=1.0*, *loss_prob=0.0*, *test_delay=0*)
Method to test people

Parameters

- **inds** – indices of who to test
- **test_sensitivity** (*float*) – probability of a true positive
- **loss_prob** (*float*) – probability of loss to follow-up
- **test_delay** (*int*) – number of days before test results are ready

Returns Whether or not this person tested positive

quarantine()
Quarantine selected people starting on the current day. If a person is already quarantined, this will extend their quarantine. :param inds: indices of who to quarantine, specified by check_quar() :type inds: array

trace(*inds*, *trace_probs*, *trace_time*)
Trace the contacts of the people provided :param inds: indices of whose contacts to trace :type inds: array :param trace_probs: probability of being able to trace people at each contact layer - should have the same keys as contacts :type trace_probs: dict :param trace_time: days it'll take to trace people at each contact layer - should have the same keys as contacts :type trace_time: dict

plot(**args*, ***kwargs*)
Plot statistics of the population – age distribution, numbers of contacts, and overall weight of contacts (number of contacts multiplied by beta per layer).

class Sim(*pars=None*, *datafile=None*, *datacols=None*, *label=None*, *simfile=None*, *popfile=None*, *load_pop=False*, *save_pop=False*, ***kwargs*)
Bases: *poliosim.base.BaseSim*

The Sim class handles the running of the simulation: the number of children, number of time points, and the parameters of the simulation.

Parameters

- **pars** (*dict*) – parameters to modify from their default values
- **datafile** (*str/df*) – filename of (Excel, CSV) data file to load, or a pandas dataframe of the data
- **datacols** (*list*) – list of column names of the data to load
- **label** (*str*) – the name of the simulation (useful to distinguish in batch runs)
- **simfile** (*str*) – the filename for this simulation, if it's saved (default: creation date)
- **popfile** (*str*) – the filename to load/save the population for this simulation
- **load_pop** (*bool*) – whether to load the population from the named file
- **save_pop** (*bool*) – whether to save the population to the named file
- **kwargs** (*dict*) – passed to make_pars()

Examples:

```
sim = ps.Sim()
sim = ps.Sim(pop_size=10e3, datafile='my_data.xlsx')
```

initialize(***kwargs*)
Perform all initializations, including validating the parameters, setting the random number seed, creating the results structure, initializing the people, validating the layer parameters (which requires the people), and initializing the interventions.

Parameters **kwargs** (*dict*) – passed to init_people

init_results()
Create the main results structure. We differentiate between flows, stocks, and cumulative results The prefix "new" is used for flow variables, i.e. counting new events (infections/recoveries) on each timestep The prefix "n" is used for stock variables, i.e. counting the total number in any given state (sus/inf/rec/etc) on any particular timestep The prefix "cum" is used for cumulative variables, i.e. counting the total number that have ever been in a given state at some point in the sim

rescale()
Dynamically rescale the population – used during step()

step()
Step the simulation forward in time. Usually, the user would use sim.run() rather than calling sim.step() directly.

run(*do_plot=False*, *until=None*, *restore_pars=True*, *reset_seed=True*, *verbose=None*)
Run the simulation.

Parameters

- **do_plot** (*bool*) – whether to plot
- **until** (*int/str*) – day or date to run until
- **restore_pars** (*bool*) – whether to make a copy of the parameters before the run and restore it after, so runs are repeatable
- **reset_seed** (*bool*) – whether to reset the random number stream immediately before run
- **verbose** (*float*) – level of detail to print, e.g. 0 = no output, 0.2 = print every 5th day, 1 = print every day

Returns A pointer to the sim object (with results modified in-place)

finalize(*verbose=None*, *restore_pars=True*)
Compute final results

compute_results(*verbose=None*)
Perform final calculations on the results

compute_prev_inci()
Compute prevalence and incidence. Prevalence is the current number of infected people divided by the number of people who are alive. Incidence is the number of new infections per day divided by the susceptible population.

initialize_immunity(*people*, *sia_coverage*, *ri_coverage*, *ri_ages_years*)

prob_infection(*beta*, *current_immunities*, *doses=1000000*, *alpha=0.44*, *gamma=0.46*)

1.4.2.5 poliosim.parameters module

Set the parameters for Poliosim.

strain_map(*key=None*, *reverse=False*)
Map from strain number to string

make_pars(*set_prognoses=False*, ***kwargs*)
Create the parameters for the simulation. Typically, this function is used internally rather than called by the user; e.g. typical use would be to do sim = ps.Sim() and then inspect sim.pars, rather than calling this function directly.

Parameters

- **set_prognoses** (*bool*) – whether or not to create prognoses (else, added when the population is created)
- **kwargs** (*dict*) – any additional kwargs are interpreted as parameter names

Returns the parameters of the simulation

Return type pars (dict)

1.4.2.6 poliosim.plotting module

Core plotting functions for simulations, multisims, and scenarios.

Also includes Plotly-based plotting functions to supplement the Matplotlib based ones that are of the Sim and Scenarios objects. Intended mostly for use with the webapp.

get_sim_plots(*which='default'*)
Specify which quantities to plot; used in sim.py.

Parameters **which** (*str*) – either 'default' or 'overview'

get_scen_plots(*which='default'*)
Default scenario plots – used in run.py

plot_sim(*sim*, *to_plot=None*, *do_save=None*, *fig_path=None*, *fig_args=None*, *plot_args=None*, *scatter_args=None*, *axis_args=None*, *fill_args=None*, *legend_args=None*, *show_args=None*, *as_dates=True*, *dateformat=None*, *interval=None*, *n_cols=None*, *font_size=18*, *font_family=None*, *grid=False*, *commaticks=True*, *setylim=True*, *log_scale=False*, *colors=None*, *labels=None*, *do_show=True*, *sep_figs=False*, *fig=None*)
Plot the results of a single simulation – see Sim.plot() for documentation

plot_scens(*scens*, *to_plot=None*, *do_save=None*, *fig_path=None*, *fig_args=None*, *plot_args=None*, *scatter_args=None*, *axis_args=None*, *fill_args=None*, *legend_args=None*, *show_args=None*, *as_dates=True*, *dateformat=None*, *interval=None*, *n_cols=None*, *font_size=18*, *font_family=None*, *grid=False*, *commaticks=True*, *setylim=True*, *log_scale=False*, *colors=None*, *labels=None*, *do_show=True*, *sep_figs=False*, *fig=None*)
Plot the results of a scenario – see Scenarios.plot() for documentation

plot_result(*sim*, *key*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *scatter_args=None*, *font_size=18*, *font_family=None*, *grid=False*, *commaticks=True*, *setylim=True*, *as_dates=True*, *dateformat=None*, *interval=None*, *color=None*, *label=None*, *fig=None*, *do_show=True*, *do_save=False*, *fig_path=None*)
Plot a single result – see Sim.plot_result() for documentation

plot_compare(*df*, *log_scale=True*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *scatter_args=None*, *font_size=18*, *font_family=None*, *grid=False*, *commaticks=True*, *setylim=True*, *as_dates=True*, *dateformat=None*, *interval=None*, *color=None*, *label=None*, *fig=None*)
Plot a MultiSim comparison – see MultiSim.plot_compare() for documentation

plot_people(*people*, *bins=None*, *width=1.0*, *font_size=18*, *alpha=0.6*, *fig_args=None*, *axis_args=None*, *plot_args=None*)
Plot statistics of a population – see People.plot() for documentation

1.4.2.7 poliosim.population module

Defines functions for making the population.

make_people(*sim*, *save_pop=False*, *popfile=None*, *die=True*, *reset=False*, *verbose=None*, ***kwargs*)
Make the actual people for the simulation. Usually called via sim.initialize(), not directly by the user.

Parameters

- **sim** (Sim) – the simulation object
- **save_pop** (*bool*) – whether to save the population to disk
- **popfile** (*bool*) – if so, the filename to save to
- **die** (*bool*) – whether or not to fail if synthetic populations are requested but not available
- **reset** (*bool*) – whether to force population creation even if self.popdict/self.people exists
- **verbose** (*bool*) – level of detail to print
- **kwargs** (*dict*) – passed to make_randpop() or make_synthpop()

Returns people

Return type people (*People*)

make_randpop(*sim*, *use_age_data=True*, *use_household_data=True*, *sex_ratio=0.5*, *microstructure=False*)
Make a random population, with contacts.

This function returns a "popdict" dictionary, which has the following (required) keys:

- uid: an array of (usually consecutive) integers of length N, uniquely identifying each agent
- age: an array of floats of length N, the age in years of each agent
- sex: an array of integers of length N (not currently used, so does not have to be binary)
- contacts: list of length N listing the contacts; see make_random_contacts() for details
- layer_keys: a list of strings representing the different contact layers in the population; see make_random_contacts() for details

Parameters

- **sim** (Sim) – the simulation object
- **use_age_data** (*bool*) – whether to use location-specific age data
- **use_household_data** (*bool*) – whether to use location-specific household size data
- **sex_ratio** (*float*) – proportion of the population that is male (not currently used)
- **microstructure** (*bool*) – whether or not to use the microstructuring algorithm to group contacts

Returns a dictionary representing the population, with the following keys for a population of N agents with M contacts between them:

Return type popdict (dict)

make_random_contacts(*pop_size*, *contacts*, *overshoot=1.2*, *dispersion=None*)
Make random static contacts.

Parameters

- **pop_size** (*int*) – number of agents to create contacts between (N)
- **contacts** (*dict*) – a dictionary with one entry per layer describing the average number of contacts per person for that layer
- **overshoot** (*float*) – to avoid needing to take multiple Poisson draws
- **dispersion** (*float*) – if not None, use a negative binomial distribution with this dispersion parameter instead of Poisson to make the contacts

Returns a list of length N, where each entry is a dictionary by layer, and each dictionary entry is the UIDs of the agent's contacts layer_keys (list): a list of layer keys, which is the same as the keys of the input "contacts" dictionary

Return type contacts_list (list)

make_microstructured_contacts(*pop_size*, *contacts*)
Create microstructured contacts – i.e. for households

make_hybrid_contacts(*pop_size*, *ages*, *contacts*, *school_ages=None*, *work_ages=None*)
Create "hybrid" contacts – microstructured contacts for households and random contacts for schools and workplaces, both of which have extremely basic age structure. A combination of both make_random_contacts() and make_microstructured_contacts().

make_hybrid_contacts_mixed_community(*pop_size*, *ages*, *contacts*, *school_ages=None*, *work_ages=None*, *mix_matrix=None*, *selectors=None*)
Same as make_hybrid_contacts, but the community layer is replaced.

The contacts in the community layer are determined by the mix_matrix and the list of selectors. The mix matrix is an M rows x N columns matrix. Only the upper right triangle of the mix matrix is used, including the diagonal.

For each used entry (i,j) in the mix matrix, the i'th entry in selectors is used to determine the pool of source individuals for each contact, while the j'th entry in selectors is used to determine the pool of target individuals for each contact. Then each source individual is assigned on average mix_matrix(i,j) number of contacts from the target pool.

make_synthpop(*sim*, *generate=True*, *layer_mapping=None*, ***kwargs*)
Make a population using SynthPops, including contacts. Usually called automatically, but can also be called manually.

Parameters

- **sim** (Sim) – a Poliosim simulation object
- **generate** (*bool*) – whether or not to generate a new population (otherwise, tries to load a pre-generated one)
- **layer_mapping** (*dict*) – a custom mapping from SynthPops layers to Poliosim layers
- **kwars** (*dict*) – passed to sp.make_population()

Example:

```
sim = ps.Sim(pop_type='synthpops')
sim.popdict = ps.make_synthpop(sim)
sim.run()
```

1.4.2.8 poliosim.run module

Functions and classes for running multiple Poliosim runs.

class MultiSim(*sims=None*, *base_sim=None*, *quantiles=None*, *initialize=False*, *label=None*, ***kwargs*)

Bases: sciris.sc_utils.prettyobj

Class for running multiple copies of a simulation. The parameter n_runs controls how many copies of the simulation there will be, if a list of sims is not provided.

Parameters

- **sims** (*Sim/list*) – a single sim or a list of sims
- **base_sim** (Sim) – the sim used for shared properties; if not supplied, the first of the sims provided
- **quantiles** (*dict*) – the quantiles to use with reduce(), e.g. [0.1, 0.9] or {'low : '0.1, 'high' : 0.9}
- **initialize** (*bool*) – whether or not to initialize the sims (otherwise, initialize them during run)
- **kwargs** (*dict*) – stored in run_args and passed to run()

Returns a MultiSim object

Return type msim

Examples:

```
sim = ps.Sim() # Create the sim
msim = ps.MultiSim(sim, n_runs=5) # Create the multisim
msim.run() # Run them in parallel
msim.combine() # Combine into one sim
msim.plot() # Plot results

sim = ps.Sim() # Create the sim
msim = ps.MultiSim(sim, n_runs=11, noise=0.1, keep_people=True) # Set up a multisim␣
↪with noise
msim.run() # Run
msim.reduce() # Compute statistics
msim.plot() # Plot

sims = [ps.Sim(beta=0.015*(1+0.02*i)) for i in range(5)] # Create sims
```

(continues on next page)

(continued from previous page)

```
for sim in sims: sim.run() # Run sims in serial
msim = ps.MultiSim(sims) # Convert to multisim
msim.plot() # Plot as single sim
```

result_keys()
Attempt to retrieve the results keys from the base sim

init_sims(***kwargs*)
Initialize the sims, but don't actually run them. Syntax is the same as MultiSim.run(). Note: in most cases you can just call run() directly, there is no need to call this separately.

Parameters **kwargs** (*dict*) – passed to multi_run()

run(*reduce=False*, *combine=False*, ***kwargs*)
Run the actual sims

Parameters

- **reduce** (*bool*) – whether or not to reduce after running (see reduce())
- **combine** (*bool*) – whether or not to combine after running (see combine(), not compatible with reduce)
- **kwargs** (*dict*) – passed to multi_run(); use run_args to pass arguments to sim.run()

Returns None (modifies MultiSim object in place)

Examples:

```
msim.run()
msim.run(run_args=dict(until='2020-0601', restore_pars=False))
```

shrink(***kwargs*)
Not to be confused with reduce(), this shrinks each sim in the msim; see sim.shrink() for more information.

Parameters **kwargs** (*dict*) – passed to sim.shrink() for each sim

reset()
Undo a combine() or reduce() by resetting the base sim, which, and results

reduce(*quantiles=None*, *output=False*)
Combine multiple sims into a single sim with scaled results

mean(*bounds=None*, ***kwargs*)
Alias for reduce(use_mean=True). See reduce() for full description.

Parameters

- **bounds** (*float*) – multiplier on the standard deviation for the upper and lower bounds (default, 2)
- **kwargs** (*dict*) – passed to reduce()

median(*quantiles=None*, ***kwargs*)
Alias for reduce(use_mean=False). See reduce() for full description.

Parameters

- **quantiles** (*list or dict*) – upper and lower quantiles (default, 0.1 and 0.9)
- **kwargs** (*dict*) – passed to reduce()

combine(*output=False*)
Combine multiple sims into a single sim with scaled results.

Example:

```
msim = cv.MultiSim(cv.Sim())
msim.run()
msim.combine()
msim.summarize()
```

compare(*t=None*, *sim_inds=None*, *output=False*, *do_plot=False*, ***kwargs*)
Create a dataframe compare sims at a single point in time.

Parameters

- **t** (*int/str*) – the day (or date) to do the comparison; default, the end
- **sim_inds** (*list*) – list of integers of which sims to include (default: all)
- **output** (*bool*) – whether or not to return the comparison as a dataframe
- **do_plot** (*bool*) – whether or not to plot the comparison (see also plot_compare())
- **kwargs** (*dict*) – passed to plot_compare()

Returns a dataframe comparison

Return type df (dataframe)

plot(*to_plot=None*, *inds=None*, *plot_sims=False*, *color_by_sim=None*, *max_sims=5*, *colors=None*, *labels=None*, *alpha_range=None*, *plot_args=None*, *show_args=None*, ***kwargs*)
Plot all the sims – arguments passed to Sim.plot(). The behavior depends on whether or not combine() or reduce() has been called. If so, this function by default plots only the combined/reduced sim (which you can override with plot_sims=True). Otherwise, it plots a separate line for each sim.

Note that this function is complex because it aims to capture the flexibility of both sim.plot() and scens.plot(). By default, if combine() or reduce() has been used, it will resemble sim.plot(); otherwise, it will resemble scens.plot(). This can be changed via color_by_sim, together with the other options.

Parameters

- **to_plot** (*list*) – list or dict of which results to plot; see cv.get_default_plots() for structure
- **inds** (*list*) – if not combined or reduced, the indices of the simulations to plot (if None, plot all)
- **plot_sims** (*bool*) – whether to plot individual sims, even if combine() or reduce() has been used
- **color_by_sim** (*bool*) – if True, set colors based on the simulation type; otherwise, color by result type; True implies a scenario-style plotting, False implies sim-style plotting
- **max_sims** (*int*) – maximum number of sims to use with color-by-sim; can be overridden by other options
- **colors** (*list*) – if supplied, override default colors for color_by_sim
- **labels** (*list*) – if supplied, override default labels for color_by_sim
- **alpha_range** (*list*) – a 2-element list/tuple/array providing the range of alpha values to use to distinguish the lines
- **plot_args** (*dict*) – passed to sim.plot()

- **show_args** (*dict*) – passed to sim.plot()
- **kwargs** (*dict*) – passed to sim.plot()

Returns Figure handle

Return type fig

Examples:

```
sim = ps.Sim()
msim = ps.MultiSim(sim)
msim.run()
msim.plot() # Plots individual sims
msim.reduce()
msim.plot() # Plots the combined sim
```

plot_result(*key*, *colors=None*, *labels=None*, **args*, ***kwargs*)
Convenience method for plotting – arguments passed to Sim.plot_result()

plot_compare(*t=-1*, *sim_inds=None*, *log_scale=True*, ***kwargs*)
Plot a comparison between sims, using bars to show different values for each result.

Parameters

- **t** (*int*) – index of results, passed to compare()
- **sim_inds** (*list*) – which sims to include, passed to compare()
- **log_scale** (*bool*) – whether to plot with a logarithmic x-axis
- **kwargs** (*dict*) – standard plotting arguments, see Sim.plot() for explanation

Returns the figure handle

Return type fig (figure)

save(*filename=None*, *keep_people=False*, ***kwargs*)
Save to disk as a gzipped pickle. Load with ps.load(filename) or ps.MultiSim.load(filename).

Parameters

- **filename** (*str*) – the name or path of the file to save to; if None, uses default
- **keep_people** (*bool*) – whether or not to store the population in the Sim objects (NB, very large)
- **kwargs** (*dict*) – passed to makefilepath()

Returns the validated absolute path to the saved file

Return type scenfile (str)

Example:

```
msim.save() # Saves to an .msim file
```

static load(*msimfile*, **args*, ***kwargs*)
Load from disk from a gzipped pickle.

Parameters

- **msimfile** (*str*) – the name or path of the file to load from
- **kwargs** – passed to ps.load()

Returns the loaded MultiSim object

Return type msim (*MultiSim*)

Example:

```
msim = ps.MultiSim.load('my-multisim.msim')
```

static merge(**args*, *base=False*)

Convenience method for merging two MultiSim objects.

Parameters

- **args** (MultiSim) – the MultiSims to merge (either a list, or separate)
- **base** (*bool*) – if True, make a new list of sims from the multisim's two base sims; otherwise, merge the multisim's lists of sims

Returns a new MultiSim object

Return type msim (*MultiSim*)

Examples:

mm1 = ps.MultiSim.merge(msim1, msim2, base=True) mm2 = ps.MultiSim.merge([m1, m2, m3, m4], base=False)

split(*inds=None*, *chunks=None*)

Convenience method for splitting one MultiSim into several. You can specify either individual indices of simulations to extract, via inds, or consecutive chunks of indices, via chunks. If this function is called on a merged MultiSim, the chunks can be retrieved automatically and no arguments are necessary.

Parameters

- **inds** (*list*) – a list of lists of indices, with each list turned into a MultiSim
- **chunks** (*int or list*) – if an int, split the MultiSim into chunks of that length; if a list return chunks of that many sims

Returns A list of MultiSim objects

Examples:

```
m1 = ps.MultiSim(ps.Sim(label='sim1'), initialize=True)
m2 = ps.MultiSim(ps.Sim(label='sim2'), initialize=True)
m3 = ps.MultiSim.merge(m1, m2)
m3.run()
m1b, m2b = m3.split()

msim = ps.MultiSim(ps.Sim(), n_runs=6)
msim.run()
m1, m2 = msim.split(inds=[[0,2,4], [1,3,5]])
mlist1 = msim.split(chunks=[2,4]) # Equivalent to inds=[[0,1], [2,3,4,5]]
mlist2 = msim.split(chunks=3) # Equivalent to inds=[[0,1,2], [3,4,5]]
```

summarize(*output=False*)

Print a brief summary of the MultiSim

single_run(*sim*, *ind=0*, *reseed=True*, *noise=0.0*, *noisepar=None*, *keep_people=False*, *run_args=None*, *sim_args=None*, *verbose=None*, *do_run=True*, ***kwargs*)

Convenience function to perform a single simulation run. Mostly used for parallelization, but can also be used directly.

Parameters

- **sim** (Sim) – the sim instance to be run
- **ind** (*int*) – the index of this sim
- **reseed** (*bool*) – whether or not to generate a fresh seed for each run
- **noise** (*float*) – the amount of noise to add to each run
- **noisepar** (*str*) – the name of the parameter to add noise to
- **keep_people** (*bool*) – whether to keep the people after the sim run
- **run_args** (*dict*) – arguments passed to sim.run()
- **sim_args** (*dict*) – extra parameters to pass to the sim, e.g. 'n_infected'
- **verbose** (*int*) – detail to print
- **do_run** (*bool*) – whether to actually run the sim (if not, just initialize it)
- **kwargs** (*dict*) – also passed to the sim

Returns a single sim object with results

Return type sim (*Sim*)

Example:

```
import poliosim as ps
sim = ps.Sim() # Create a default simulation
sim = ps.single_run(sim) # Run it, equivalent(ish) to sim.run()
```

multi_run(*sim*, *n_runs=4*, *reseed=True*, *noise=0.0*, *noisepar=None*, *iterpars=None*, *combine=False*, *keep_people=None*, *run_args=None*, *sim_args=None*, *par_args=None*, *do_run=True*, *parallel=True*, *n_cpus=None*, *verbose=None*, ***kwargs*)

For running multiple runs in parallel. If the first argument is a list of sims, exactly these will be run and most other arguments will be ignored.

Parameters

- **sim** (Sim) – the sim instance to be run, or a list of sims.
- **n_runs** (*int*) – the number of parallel runs
- **reseed** (*bool*) – whether or not to generate a fresh seed for each run
- **noise** (*float*) – the amount of noise to add to each run
- **noisepar** (*str*) – the name of the parameter to add noise to
- **iterpars** (*dict*) – any other parameters to iterate over the runs; see sc.parallelize() for syntax
- **combine** (*bool*) – whether or not to combine all results into one sim, rather than return multiple sim objects
- **keep_people** (*bool*) – whether to keep the people after the sim run (default false)
- **run_args** (*dict*) – arguments passed to sim.run()
- **sim_args** (*dict*) – extra parameters to pass to the sim
- **par_args** (*dict*) – arguments passed to sc.parallelize()
- **do_run** (*bool*) – whether to actually run the sim (if not, just initialize it)

- **parallel** (*bool*) – whether to run in parallel using multiprocessing (else, just run in a loop)
- **n_cpus** (*int*) – the number of CPUs to run on (if blank, set automatically; otherwise, passed to par_args)
- **verbose** (*int*) – detail to print
- **kwargs** (*dict*) – also passed to the sim

Returns If combine is True, a single sim object with the combined results from each sim. Otherwise, a list of sim objects (default).

Example:

```
import poliosim as ps
sim = ps.Sim()
sims = ps.multi_run(sim, n_runs=6, noise=0.2)
```

run_configs(*configs*, *do_run=True*, *do_save=None*, *save_summary=False*, ***kwargs*)
Helper function to run a list of dictionaries with configuration settings.

Example:

```
TBC
```

run_sims(*config=None*, *keep_people=False*, ***kwargs*)
Main API for running sims

create_sim(*config=None*, ***kwargs*)
Function for creating a sim – should be merged with run_sims

1.4.2.9 poliosim.settings module

Define options for Poliosim, mostly plotting and Numba options. All options should be set using set(), e.g.:

```
ps.options.set(font_size=18)
```

To reset default options, use:

```
ps.options.set('default')
```

1.4.2.10 poliosim.utils module

Numerical utilities for running Poliosim

sample(*dist=None*, *par1=None*, *par2=None*, *size=None*, ***kwargs*)
Draw a sample from the distribution specified by the input.

Parameters

- **dist** (*str*) – the distribution to sample from
- **par1** (*float*) – the "main" distribution parameter (e.g. mean)
- **par2** (*float*) – the "secondary" distribution parameter (e.g. std)
- **size** (*int*) – the number of samples (default=1)
- **kwargs** (*dict*) – passed to individual sampling functions

Returns A length N array of samples

Examples:

```
sample() # returns Unif(0,1)
sample(dist='normal', par1=3, par2=0.5) # returns Normal(=3, =0.5)
```

Notes

Lognormal distributions are parameterized with reference to the underlying normal distribution (see: https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.lognormal.html), but this function assumes the user wants to specify the mean and variance of the lognormal distribution.

set_seed(*seed=None*)

Reset the random seed – complicated because of Numba, which requires special syntax to reset the seed. This function also resets Python's built-in random number generated.

Parameters **seed** (*int*) – the random seed

n_binomial(*prob*, *n*)

Perform multiple binomial (Bernolli) trials

Parameters

- **prob** (*float*) – probability of each trial succeeding
- **n** (*int*) – number of trials (size of array)

Returns Boolean array of which trials succeeded

Example:

```
outcomes = ps.n_binomial(0.5, 100) # Perform 100 coin-flips
```

binomial_filter(*prob*, *arr*)

Binomial "filter" – the same as n_binomial, except return the elements of arr that succeeded.

Parameters

- **prob** (*float*) – probability of each trial succeeding
- **arr** (*array*) – the array to be filtered

Returns Subset of array for which trials succeeded

Example:

```
inds = ps.binomial_filter(0.5, np.arange(20)**2) # Return which values out of the␣
↪(arbitrary) array passed the coin flip
```

binomial_arr(*prob_arr*)

Binomial (Bernoulli) trials each with different probabilities.

Parameters **prob_arr** (*array*) – array of probabilities

Returns Boolean array of which trials on the input array succeeded

Example:

```
outcomes = ps.binomial_arr([0.1, 0.1, 0.2, 0.2, 0.8, 0.8]) # Perform 6 trials with␣
↪different probabilities
```

n_multinomial(*probs*, *n*)
An array of multinomial trials.

Parameters

- **probs** (*array*) – probability of each outcome, which usually should sum to 1
- **n** (*int*) – number of trials

Returns Array of integer outcomes

Example:

```
outcomes = ps.n_multinomial(np.ones(6)/6.0, 50)+1 # Return 50 die-rolls
```

poisson(*rate*)
A Poisson trial.

Parameters **rate** (*float*) – the rate of the Poisson process

Example:

```
outcome = ps.poisson(100) # Single Poisson trial with mean 100
```

n_poisson(*rate*, *n*)
An array of Poisson trials.

Parameters

- **rate** (*float*) – the rate of the Poisson process (mean)
- **n** (*int*) – number of trials

Example:

```
outcomes = ps.n_poisson(100, 20) # 20 poisson trials with mean 100
```

n_neg_binomial(*rate*, *dispersion*, *n*, *step=1*)
An array of negative binomial trials; with dispersion = ∞, converges to Poisson.

Parameters

- **rate** (*float*) – the rate of the process (mean, same as Poisson)
- **dispersion** (*float*) – amount of dispersion: 0 = infinite, 1 = std is equal to mean, ∞ = Poisson
- **n** (*int*) – number of trials
- **step** (*float*) – the step size to use if non-integer outputs are desired

Example:

```
outcomes = ps.n_neg_binomial(100, 1, 20) # 20 negative binomial trials with mean␣
↪100 and dispersion equal to mean
```

choose(*max_n*, *n*)
Choose a subset of items (e.g., people) without replacement.

Parameters

- **max_n** (*int*) – the total number of items
- **n** (*int*) – the number of items to choose

Example:

```
choices = ps.choose(5, 2) # choose 2 out of 5 people with equal probability␣
↪(without repeats)
```

choose_r(*max_n*, *n*)

Choose a subset of items (e.g., people), with replacement.

Parameters

- **max_n** (*int*) – the total number of items
- **n** (*int*) – the number of items to choose

Example:

```
choices = ps.choose_r(5, 10) # choose 10 out of 5 people with equal probability␣
↪(with repeats)
```

choose_w(*probs*, *n*, *unique=True*)

Choose n items (e.g. people), each with a probability from the distribution probs.

Parameters

- **probs** (*array*) – list of probabilities, should sum to 1
- **n** (*int*) – number of samples to choose
- **unique** (*bool*) – whether or not to ensure unique indices

Example:

```
choices = ps.choose_w([0.2, 0.5, 0.1, 0.1, 0.1], 2) # choose 2 out of 5 people with␣
↪nonequal probability.
```

true(*arr*)

Returns the indices of the values of the array that are true: just an alias for arr.nonzero()[0].

Parameters **arr** (*array*) – any array

Example:

```
inds = ps.true(np.array([1,0,0,1,1,0,1]))
```

false(*arr*)

Returns the indices of the values of the array that are false.

Parameters **arr** (*array*) – any array

Example:

```
inds = ps.false(np.array([1,0,0,1,1,0,1]))
```

defined(*arr*)

Returns the indices of the values of the array that are not-nan.

Parameters **arr** (*array*) – any array

Example:

```
inds = ps.defined(np.array([1,np.nan,0,np.nan,1,0,1]))
```

undefined(*arr*)
Returns the indices of the values of the array that are not-nan.

Parameters **arr** (*array*) – any array

Example:

```
inds = ps.defined(np.array([1,np.nan,0,np.nan,1,0,1]))
```

itrue(*arr*, *inds*)
Returns the indices that are true in the array – name is short for indices[true]

Parameters

- **arr** (*array*) – a Boolean array, used as a filter
- **inds** (*array*) – any other array (usually, an array of indices) of the same size

Example:

```
inds = ps.itrue(np.array([True,False,True,True]), inds=np.array([5,22,47,93]))
```

ifalse(*arr*, *inds*)
Returns the indices that are true in the array – name is short for indices[false]

Parameters

- **arr** (*array*) – a Boolean array, used as a filter
- **inds** (*array*) – any other array (usually, an array of indices) of the same size

Example:

```
inds = ps.ifalse(np.array([True,False,True,True]), inds=np.array([5,22,47,93]))
```

idefined(*arr*, *inds*)
Returns the indices that are true in the array – name is short for indices[defined]

Parameters

- **arr** (*array*) – any array, used as a filter
- **inds** (*array*) – any other array (usually, an array of indices) of the same size

Example:

```
inds = ps.idefined(np.array([3,np.nan,np.nan,4]), inds=np.array([5,22,47,93]))
```

itruei(*arr*, *inds*)
Returns the indices that are true in the array – name is short for indices[true[indices]]

Parameters

- **arr** (*array*) – a Boolean array, used as a filter
- **inds** (*array*) – an array of indices for the original array

Example:

```
inds = ps.itruei(np.array([True,False,True,True,False,False,True,False]), inds=np.
↪array([0,1,3,5]))
```

ifalsei(*arr*, *inds*)
Returns the indices that are false in the array – name is short for indices[false[indices]]

Parameters

- **arr** (*array*) – a Boolean array, used as a filter
- **inds** (*array*) – an array of indices for the original array

Example:

```
inds = ps.ifalsei(np.array([True,False,True,True,False,False,True,False]), inds=np.
↪array([0,1,3,5]))
```

idefinedi(*arr*, *inds*)

Returns the indices that are defined in the array – name is short for indices[defined[indices]]

Parameters

- **arr** (*array*) – any array, used as a filter
- **inds** (*array*) – an array of indices for the original array

Example:

```
inds = ps.idefinedi(np.array([4,np.nan,0,np.nan,np.nan,4,7,4,np.nan]), inds=np.
↪array([0,1,3,5]))
```

load(*filename=None*, *folder=None*, *verbose=False*, *die=None*, *remapping=None*, *method='pickle'*, ***kwargs*)

Load a file that has been saved as a gzipped pickle file, e.g. by `sc.saveobj()`. Accepts either a filename (standard usage) or a file object as the first argument. Note that `loadobj()`/`load()` are aliases of each other.

Note: be careful when loading pickle files, since a malicious pickle can be used to execute arbitrary code.

When a pickle file is loaded, Python imports any modules that are referenced in it. This is a problem if module has been renamed. In this case, you can use the `remapping` argument to point to the new modules or classes.

Parameters

- **filename** (*str/Path*) – the filename (or full path) to load
- **folder** (*str/Path*) – the folder
- **verbose** (*bool*) – print details
- **die** (*bool*) – whether to raise an exception if errors are encountered (otherwise, load as much as possible)
- **remapping** (*dict*) – way of mapping old/unavailable module names to new
- **method** (*str*) – method for loading (usually pickle or dill)
- **kwargs** (*dict*) – passed to pickle.loads()/dill.loads()

Examples:

```
obj = sc.loadobj('myfile.obj') # Standard usage
old = sc.loadobj('my-old-file.obj', method='dill', ignore=True) # Load classes from␣
↪saved files
old = sc.loadobj('my-old-file.obj', remapping={'foo.Bar':cat.Mat}) # If loading a␣
↪saved object containing a reference to foo.Bar that is now cat.Mat
old = sc.loadobj('my-old-file.obj', remapping={'foo.Bar':('cat', 'Mat')}) #␣
↪Equivalent to the above
```

New in version 1.1.0: "remapping" argument New in version 1.2.2: ability to load non-gzipped pickles; support for dill; arguments passed to loader

save(*filename=None*, *obj=None*, *compresslevel=5*, *verbose=0*, *folder=None*, *method='pickle'*, *die=True*, **args*, ***kwargs*)

Save an object to file as a gzipped pickle – use compression 5 by default, since more is much slower but not much smaller. Once saved, can be loaded with sc.loadobj(). Note that saveobj()/save() are identical.

Parameters

- **filename** (*str or Path*) – the filename to save to; if str, passed to sc.makefilepath()
- **obj** (*literally anything*) – the object to save
- **compresslevel** (*int*) – the level of gzip compression
- **verbose** (*int*) – detail to print
- **folder** (*str*) – passed to sc.makefilepath()
- **method** (*str*) – whether to use pickle (default) or dill
- **die** (*bool*) – whether to fail if no object is provided
- **args** (*list*) – passed to pickle.dumps()
- **kwargs** (*dict*) – passed to pickle.dumps()

Example:

```
myobj = ['this', 'is', 'a', 'weird', {'object':44}]
sc.saveobj('myfile.obj', myobj)
sc.saveobj('myfile.obj', myobj, method='dill') # Use dill instead, to save custom␣
↪classes as well
```

New in version 1.1.1: removed Python 2 support. New in version 1.2.2: automatic swapping of arguments if order is incorrect; correct passing of arguments

date(*obj*, **args*, *start_date=None*, *readformat=None*, *outformat=None*, *as_date=True*, ***kwargs*)

Convert any reasonable object – a string, integer, or datetime object, or list/array of any of those – to a date object. To convert an integer to a date, you must supply a start date.

Caution: while this function and readdate() are similar, and indeed this function calls readdate() if the input is a string, in this function an integer is treated as a number of days from start_date, while for readdate() it is treated as a timestamp in seconds. To change

Parameters

- **obj** (*str, int, date, datetime, list, array*) – the object to convert
- **args** (*str, int, date, datetime*) – additional objects to convert
- **start_date** (*str, date, datetime*) – the starting date, if an integer is supplied
- **readformat** (*str/list*) – the format to read the date in; passed to sc.readdate()
- **outformat** (*str*) – the format to output the date in, if returning a string
- **as_date** (*bool*) – whether to return as a datetime date instead of a string

Returns either a single date object, or a list of them (matching input data type where possible)

Return type dates (date or list)

Examples:

```
sc.date('2020-04-05') # Returns datetime.date(2020, 4, 5)
sc.date([35,36,37], start_date='2020-01-01', as_date=False) # Returns ['2020-02-05',
↪'2020-02-06', '2020-02-07']
sc.date(1923288822, readformat='posix') # Interpret as a POSIX timestamp
```

New in version 1.0.0. New in version 1.2.2: "readformat" argument; renamed "dateformat" to "outformat"

day(*obj*, **args*, *start_date=None*, ***kwargs*)

Convert a string, date/datetime object, or int to a day (int), the number of days since the start day. See also sc.date() and sc.daydiff(). If a start day is not supplied, it returns the number of days into the current year.

Parameters

- **obj** (*str, date, int, list, array*) – convert any of these objects to a day relative to the start day
- **args** (*list*) – additional days
- **start_date** (*str or date*) – the start day; if none is supplied, return days since (supplied year)-01-01.

Returns the day(s) in simulation time (matching input data type where possible)

Return type days (int or list)

Examples:

```
sc.day(sc.now()) # Returns how many days into the year we are
sc.day(['2021-01-21', '2024-04-04'], start_date='2022-02-22') # Days can be␣
↪positive or negative
```

New in version 1.0.0. New in version 1.2.2: renamed "start_day" to "start_date"

daydiff(**args*)

Convenience function to find the difference between two or more days. With only one argument, calculate days since 2020-01-01.

Examples:

```
diff  = sc.daydiff('2020-03-20', '2020-04-05') # Returns 16
diffs = sc.daydiff('2020-03-20', '2020-04-05', '2020-05-01') # Returns [16, 26]
```

New in version 1.0.0.

date_range(*start_date*, *end_date*, *inclusive=True*, *as_date=False*, *dateformat=None*)

Return a list of dates from the start date to the end date. To convert a list of days (as integers) to dates, use sc.date() instead.

Parameters

- **start_date** (*int/str/date*) – the starting date, in any format
- **end_date** (*int/str/date*) – the end date, in any format
- **inclusive** (*bool*) – if True (default), return to end_date inclusive; otherwise, stop the day before
- **as_date** (*bool*) – if True, return a list of datetime.date objects instead of strings
- **dateformat** (*str*) – passed to date()

Example:

```
dates = sc.daterange('2020-03-01', '2020-04-04')
```

New in version 1.0.0.

load_data(*datafile*, *columns=None*, *calculate=True*, *check_date=True*, *verbose=True*, ***kwargs*)

Load data for comparing to the model output, either from file or from a dataframe.

Parameters

- **datafile** (*str or df*) – if a string, the name of the file to load (either Excel or CSV); if a dataframe, use directly
- **columns** (*list*) – list of column names (otherwise, load all)
- **calculate** (*bool*) – whether to calculate cumulative values from daily counts
- **check_date** (*bool*) – whether to check that a 'date' column is present
- **kwargs** (*dict*) – passed to pd.read_excel()

Returns pandas dataframe of the loaded data

Return type data (dataframe)

savefig(*filename=None*, *comments=None*, ***kwargs*)

Wrapper for Matplotlib's savefig() function which automatically stores poliosim metadata in the figure. By default, saves

Parameters

- **filename** (*str*) – name of the file to save to (default, timestamp)
- **comments** (*str*) – additional metadata to save to the figure
- **kwargs** (*dict*) – passed to savefig()

Example:

```
ps.Sim().run(do_plot=True)
filename = ps.savefig()
```

get_png_metadata(*filename*, *output=False*)

Read metadata from a PNG file. For use with images saved with ps.savefig(). Requires pillow, an optional dependency.

Parameters **filename** (*str*) – the name of the file to load the data from

Example:

```
ps.Sim().run(do_plot=True)
ps.savefig('poliosim.png')
ps.get_png_metadata('poliosim.png')
```

git_info(*filename=None*, *check=False*, *comments=None*, *old_info=None*, *die=False*, *indent=2*, *verbose=True*, ***kwargs*)

Get current git information and optionally write it to disk. Simplest usage is ps.git_info(__file__)

Parameters

- **filename** (*str*) – name of the file to write to or read from
- **check** (*bool*) – whether or not to compare two git versions
- **comments** (*str/dict*) – additional comments to include in the file

- **old_info** (*dict*) – dictionary of information to check against
- **die** (*bool*) – whether or not to raise an exception if the check fails
- **indent** (*int*) – how many indents to use when writing the file to disk
- **verbose** (*bool*) – detail to print
- **kwargs** (*dict*) – passed to loadjson (if check=True) or loadjson (if check=False)

Examples:

```
ps.git_info() # Return information
ps.git_info(__file__) # Writes to disk
ps.git_info('poliosim_version.gitinfo') # Writes to disk
ps.git_info('poliosim_version.gitinfo', check=True) # Checks that current version␣
↪matches saved file
```

check_version(*expected*, *die=False*, *verbose=True*, ***kwargs*)

Get current git information and optionally write it to disk.

Parameters

- **expected** (*str*) – expected version information
- **die** (*bool*) – whether or not to raise an exception if the check fails

check_save_version(*expected=None*, *filename=None*, *die=False*, *verbose=True*, ***kwargs*)

A convenience function that bundles check_version with git_info and saves automatically to disk from the calling file. The idea is to put this at the top of an analysis script, and commit the resulting file, to keep track of which version of poliosim was used.

Parameters

- **expected** (*str*) – expected version information
- **filename** (*str*) – file to save to; if None, guess based on current file name
- **kwargs** (*dict*) – passed to git_info()

Examples:

```
ps.check_save_version()
ps.check_save_version('1.3.2', filename='script.gitinfo', comments='This is the␣
↪main analysis script')
```

compute_gof(*actual*, *predicted*, *normalize=True*, *use_frac=False*, *use_squared=False*, *as_scalar='none'*, *eps=1e-09*, *skestimator=None*, ***kwargs*)

Calculate the goodness of fit. By default use normalized absolute error, but highly customizable. For example, mean squared error is equivalent to setting normalize=False, use_squared=True, as_scalar='mean'.

Parameters

- **actual** (*arr*) – array of actual (data) points
- **predicted** (*arr*) – corresponding array of predicted (model) points
- **normalize** (*bool*) – whether to divide the values by the largest value in either series
- **use_frac** (*bool*) – convert to fractional mismatches rather than absolute
- **use_squared** (*bool*) – square the mismatches
- **as_scalar** (*str*) – return as a scalar instead of a time series: choices are sum, mean, median

- **eps** (*float*) – to avoid divide-by-zero
- **skestimator** (*str*) – if provided, use this scikit-learn estimator instead
- **kwargs** (*dict*) – passed to the scikit-learn estimator

Returns array of goodness-of-fit values, or a single value if as_scalar is True

Return type gofs (arr)

Examples:

```
x1 = np.cumsum(np.random.random(100))
x2 = np.cumsum(np.random.random(100))

e1 = compute_gof(x1, x2) # Default, normalized absolute error
e2 = compute_gof(x1, x2, normalize=False, use_frac=False) # Fractional error
e3 = compute_gof(x1, x2, normalize=False, use_squared=True, as_scalar='mean') #␣
↪Mean squared error
e4 = compute_gof(x1, x2, estimator='mean_squared_error') # Scikit-learn's MSE method
e5 = compute_gof(x1, x2, as_scalar='median') # Normalized median absolute error --␣
↪highly robust
```

diff_sims(*sim1*, *sim2*, *skip_key_diffs=False*, *output=False*, *die=False*)

Compute the difference of the summaries of two simulations, and print any values which differ.

Parameters

- **sim1** (*sim/dict*) – either a simulation object or the sim.summary dictionary
- **sim2** (*sim/dict*) – ditto
- **skip_key_diffs** (*bool*) – whether to skip keys that don't match between sims
- **output** (*bool*) – whether to return the output as a string (otherwise print)
- **die** (*bool*) – whether to raise an exception if the sims don't match
- **require_run** (*bool*) – require that the simulations have been run

Example:

```
s1 = cv.Sim(beta=0.01)
s2 = cv.Sim(beta=0.02)
s1.run()
s2.run()
cv.diff_sims(s1, s2)
```

1.4.2.11 poliosim.version module

PYTHON MODULE INDEX

p

INDEX

E

F

G

I

K

L

M

N

P

Q

R

S

T

U

V