fpsim

Institute for Disease Modeling

May 22, 2022

CONTENTS

This repository contains the code for both the family planning model, FPsim, as well as the analysis scripts for performing analyses.

- FPsim, in the folder `fpsim`, is a standalone Python library for performing family planning analyses.
- Scripts that use FPsim to run analyses are in the `fp_analyses` folder.
- FPsim tests are in the `tests` folder.
- Most other folders are preserved for archival purposes and may be deleted.
- This repository is currently under development. Eventually, FPsim and `fp_analyses` will be split into a separate repositories.

CHAPTER

ONE

INSTALLATION

Run `pip install -e .` to install these packages and their required dependencies. This will make both `fpsim` and `fp_analyses` available on the Python path.

CHAPTER

TWO

DOCUMENTATION

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

CHAPTER

THREE

CONTRIBUTING

Style guide

Please follow the starsim style guide at: https://github.com/amath-idm/styleguide

Issues

- 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.
- Label issues you are currently working on with `in progress` for tracking purposes - and to avoid accidental replication of work.
- High priority issues are organized from top (most urgent) to bottom (least urgent) and can be labelled with `urgent` or `blocking` 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!)
- The Hydra Head Effect: Often when you solve one issue, two more pop up in its place. When this happens, close the original issue and start new issues (linked) to be triaged.
- If your issue has more than two distinct tasks associated with it, please include a check list in the text, so that we can track which components of the issue have been resolved and which need to be supported.
- If your issue is a bug that was not caught by test, and includes a specific expected value that can be hard-checked, please either include or request a test patch so that a test fails due to the bug

Pull Requests

- ALL PRs should be linked to at least one issue. As above, if you're working on a PR and there's no issue associated with it, you can create an issue. However, before doing so, ask yourself if it really needs to be done.
- All PRs should have another person assigned for review. If assigned to more than one person, use the comment section to assign an issue owner/main reviewer. Use your best judgement here, as roles shift, but in general:
  - @MOBrien-IDM as FPsim lead (approval required to merge)
  - Anyone you've worked with on this issue 1:1
  - @cliffckerr to ensure new feature performance and compatibility with FPsim
  - @mzimmermann-IDM for subject matter expertise, economic and empowerment questions, questions about modeling best practices
  - @avictorious for questions about historical FPsim decisions and subject matter expertise
  - @SBuxton-IDM for embedded engineer support, public repo tasks, and testing
- Keep PRs as small as possible: e.g., one issue, one PR. Small PRs are easier to review and merge.

- 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!
- 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.
- Please create a draft PR on an active branch as soon as you're ready. Be generous in creating draft PRs. It helps with transparency and allows for quicker support if you run into a problem.
- Make sure tests pass on your PR. If they don't, mark the PR as draft until they do.
- Even if your work isn't ready for a PR, push it regularly. A guiding principle is to commit every few minutes and push to your branch every 1-2 hours.
- Every PR that adds a new feature or new functionality which can be hard-checked (so, excluding plotting functionality etc.) should include a corresponding unittest

Testing

- Every time a new feature is added, the developer should develop a unittest which checks the basic implementation of the feature
- A unittest is simply a function starting with "test" that implements a feature as succinctly as possibly, and checks the expected output with an assertion
- If you're having trouble starting a unittest feel free to look at some examples here
- Some test suites organize the tests into a class with a configuration function called `setUp()`. After implementing a unittest in such a class you may want to take advantage of the shared assets defined in `setUp()` to minimize the number of lines of code in your test.
- The new unittest should follow style guidelines laid out in the starsim style guide
- The new test should contain a docstring that details what is being tested, how it is tested (what it's being checked against), and the expected value
- The test should display error message information that is sufficient to create a bug report (summary, expected value, and actual value)

3.1 API reference

3.1.1 fpsim package

Subpackages

fpsim.locations package

Submodules

fpsim.locations.senegal module

Set the parameters for FPsim, specifically for Senegal.

fpsim.locations.senegal.**scalar_pars**()

fpsim.locations.senegal.**data2interp**(*data*, *ages*, *normalize=False*)
Convert unevenly spaced data into an even spline interpolation

fpsim.locations.senegal.**age_pyramid**()
Starting age bin, male population, female population

fpsim.locations.senegal.**age_mortality**(*bound*)
Age-dependent mortality rates, Senegal specific from 1990-1995 – see age_dependent_mortality.py in the fp_analyses repository Mortality rate trend from crude mortality rate per 1000 people: https://data.worldbank.org/indicator/SP.DYN.CDRT.IN?locations=SN

fpsim.locations.senegal.**maternal_mortality**()
Risk of maternal death assessed at each pregnancy. Data from Huchon et al. (2013) prospective study on risk of maternal death in Senegal and Mali. Maternal deaths: The annual number of female deaths from any cause related to or aggravated by pregnancy or its management (excluding accidental or incidental causes) during pregnancy and childbirth or within 42 days of termination of pregnancy, irrespective of the duration and site of the pregnancy, expressed per 100,000 live births, for a specified time period.

fpsim.locations.senegal.**infant_mortality**()
From World Bank indicators for infant morality (< 1 year) for Senegal, per 1000 live births From API_SP.DYN.IMRT.IN_DS2_en_excel_v2_1495452.numbers Adolescent increased risk of infant mortality gradient taken from Noori et al for Sub-Saharan African from 2014-2018. Odds ratios with age 23-25 as reference group: https://www.medrxiv.org/content/10.1101/2021.06.10.21258227v1

fpsim.locations.senegal.**miscarriage**()
Returns a linear interpolation of the likelihood of a miscarriage by age, taken from data from Magnus et al BMJ 2019: https://pubmed.ncbi.nlm.nih.gov/30894356/ Data to be fed into likelihood of continuing a pregnancy once initialized in model Age 0 and 5 set at 100% likelihood. Age 10 imputed to be symmetrical with probability at age 45 for a parabolic curve

fpsim.locations.senegal.**stillbirth**()
From Report of the UN Inter-agency Group for Child Mortality Estimation, 2020 https://childmortality.org/wp-content/uploads/2020/10/UN-IGME-2020-Stillbirth-Report.pdf

fpsim.locations.senegal.**female_age_fecundity**(*bound*)
Use fecundity rates from PRESTO study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5712257/ Fecundity rate assumed to be approximately linear from onset of fecundity around age 10 (average age of menses 12.5) to first data point at age 20 45-50 age bin estimated at 0.10 of fecundity of 25-27 yr olds, based on fertility rates from Senegal

fpsim.locations.senegal.**fecundity_ratio_nullip**()
Returns an array of fecundity ratios for a nulliparous woman vs a gravid woman from PRESTO study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5712257/ Approximates primary infertility and its increasing likelihood if a woman has never conceived by age

fpsim.locations.senegal.**lactational_amenorrhea**()
Returns an array of the percent of breastfeeding women by month postpartum 0-11 months who meet criteria for LAM: Exclusively breastfeeding (bf + water alone), menses have not returned. Extended out 5-11 months to better match data as those women continue to be postpartum insusceptible. From DHS Senegal calendar data

fpsim.locations.senegal.**sexual_activity**()
Returns a linear interpolation of rates of female sexual activity, defined as percentage women who have had sex within the last four weeks. From STAT Compiler DHS https://www.statcompiler.com/en/ Using indicator "Timing of sexual intercourse" Includes women who have had sex "within the last four weeks" Excludes women

who answer "never had sex", probabilities are only applied to agents who have sexually debuted Data taken from 2018 DHS, no trend over years for now Onset of sexual activity probabilities assumed to be linear from age 10 to first data point at age 15

fpsim.locations.senegal.**sexual_activity_pp**()
Returns an array of monthly likelihood of having resumed sexual activity within 0-35 months postpartum Uses DHS Senegal 2018 individual recode (postpartum (v222), months since last birth, and sexual activity within 30 days. Limited to 35 months postpartum (can use any limit you want 0-35 max) Postpartum month 0 refers to the first month after delivery

fpsim.locations.senegal.**debut_age**()
Returns an array of weighted probabilities of sexual debut by a certain age 10-45. Data taken from DHS variable v531 (imputed age of sexual debut, imputed with data from age at first union) Use sexual_debut_age_probs.py under fp_analyses/data to output for other DHS countries

fpsim.locations.senegal.**exposure_age**()
Returns an array of experimental factors to be applied to account for residual exposure to either pregnancy or live birth by age. Exposure to pregnancy will increase factor number and residual likelihood of avoiding live birth (mostly abortion, also miscarriage), will decrease factor number

fpsim.locations.senegal.**exposure_parity**()
Returns an array of experimental factors to be applied to account for residual exposure to either pregnancy or live birth by parity.

fpsim.locations.senegal.**birth_spacing_pref**()
Returns an array of birth spacing preferences by closest postpartum month. Applied to postpartum pregnancy likelihoods.

NOTE: spacing bins must be uniform!

fpsim.locations.senegal.**methods**()
Matrices to give transitional probabilities from 2018 DHS Senegal contraceptive calendar data Probabilities in this function are annual probabilities of initiating, discontinuing, continuing or switching methods.

Probabilities at postpartum month 1 are 1 month transitional probabilities for starting a method after delivery.

Probabilities at postpartum month 6 are 5 month transitional probabilities for starting or changing methods over the first 6 months postpartum.

Data from Senegal DHS contraceptive calendars, 2017 and 2018 combined

fpsim.locations.senegal.**efficacy**(*disable=False*)
From Guttmacher, fp/docs/gates_review/contraceptive-failure-rates-in-developing-world_1.pdf BTL failure rate from general published data Pooled efficacy rates for all women in this study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4970461/

fpsim.locations.senegal.**efficacy25**(*disable=False*)
From Guttmacher, fp/docs/gates_review/contraceptive-failure-rates-in-developing-world_1.pdf BTL failure rate from general published data Pooled efficacy rates for women ages 25+

fpsim.locations.senegal.**barriers**()
Reasons for nonuse – taken from DHS

fpsim.locations.senegal.**make_pars**(*configuration_file=None*, *defaults_file=None*, *bound=True*)
Take all parameters and construct into a dictionary

fpsim.locations.senegal.**validate_pars**(*pars*)
Perform internal validation checks and other housekeeping

Submodules

fpsim.analyzers module

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

class fpsim.analyzers.**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

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 fpsim.analyzers.**snapshot**(*timesteps*, **args*, *die=True*, ***kwargs*)
Bases: *fpsim.analyzers.Analyzer*

Analyzer that takes a "snapshot" of the sim.people array at specified points in time, and saves them to itself.

Parameters

- **timesteps** (*list*) – list of timesteps on which to take the snapshot
- **args** (*list*) – additional timestep(s)
- **die** (*bool*) – whether or not to raise an exception if a date is not found (default true)
- **kwargs** (*dict*) – passed to Analyzer()

Example:

```
sim = cv.Sim(analyzers=fps.snapshot('2020-04-04', '2020-04-14'))
sim.run()
snapshot = sim.pars['analyzers'][0]
people = snapshot.snapshots[0]
```

apply(*sim*)
Apply snapshot at each timestep listed in timesteps and save result at snapshot[str(timestep)]

class fpsim.analyzers.**timeseries_recorder**
Bases: *fpsim.analyzers.Analyzer*

Record every attribute in people as a timeseries.

self.**i**
The list of timesteps (ie, 0 to 261 steps).

self.**t**
The time elapsed in years given how many timesteps have passed (ie, 25.75 years).

self.**y**
The calendar year of timestep (ie, 1975.75).

self.**keys**
A list of people states excluding 'dobs'.

self.**data**
A dictionary where self.data[state][timestep] is the mean of the state at that timestep.

initialize(*sim*)
Initializes self.keys from sim.people

apply(*sim*)
Applies recorder at each timestep

plot(*x='y'*, *fig_args=None*, *pl_args=None*)
Plots time series of each state as a line graph

class fpsim.analyzers.**age_pyramids**(*bins=None*)
Bases: *fpsim.analyzers.Analyzer*

Records age pyramids for each timestep.

self.**bins**
A list of ages, default is a sequence from 0 to max_age + 1.

self.**data**
A matrix of shape (number of timesteps, number of bins - 1) containing age pyramid data.

initialize(*sim*)
Initializes bins and data with proper shapes

apply(*sim*)
Records histogram of ages of all alive individuals at a timestep such that self.data[timestep] = list of proportions where index signifies age

plot()
Plots self.data as 2D pyramid plot

plot3d()
Plots self.data as 3D pyramid plot

class fpsim.analyzers.**SimVerbose**(*pars=None*, *mother_ids=False*)
Bases: *fpsim.sim.Sim*

log_daily_totals()
Logs data for total_results and events at each timestep.

Output:

self.total_results::dict Dictionary of all individual results formatted as {timestep: attribute: [values]} keys correspond to fpsim.defaults debug_states

self.events::dict Dictionary of events correponding to self.channels formatted as {timestep: channel: [indices]}.

`save_daily_totals()`
At the end of sim run, stores total_results as either a json or feather file.

Inputs

self.to_file::bool If True, writes results to file

self.to_feather::bool If True, writes results to feather file If False, writes results to json file

Outputs: Either a json file at "sim_output/total_results.json" or a feather file for each state at "sim_output/{state}_state"

`story(index)`
Prints a story of all major events in an individual's life based on calculated SimVerbose channels, base Sim channels, and statistics calculated within the function such as year of birth of individual.

Inputs:

index::int: index of the individual, must be less than population

Outputs: printed display of each major event in the individual's life

`class fpsim.analyzers.ExperimentVerbose(pars=None, flags=None, label=None)`
Bases: *fpsim.experiment.Experiment*

`run_model(pars=None, mother_ids=False)`
Create the sim and run the model, saving total results and individual events in the process

fpsim.base module

Base classes for loading parameters and for running simulations with FP model

`class fpsim.base.ParsObj(pars, **kwargs)`
Bases: `fpsim.base.FlexPretty`

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

`update_pars(pars=None, create=False, **kwargs)`
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
- **kwargs** (*dict*) – additional parameters

`class fpsim.base.BasePeople`
Bases: `sciris.sc_utils.prettyobj`

Class for all the people in the simulation.

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

property is_female
Boolean array of everyone female

property is_male
Boolean array of everyone male

property int_age
Return ages as an integer

property ceil_age
Rounds age up to the next highest integer

property int_age_clip
Return ages as integers, clipped to maximum allowable age for pregnancy

property n
Number of people alive

property inds
Alias to self._inds to prevent accidental overwrite & increase speed

property len_inds
Alias to len(self)

property len_people
Full length of People array, ignoring filtering

plot(*fig_args=None*, *hist_args=None*)
Plot histograms of each quantity

filter(*criteria=None*, *inds=None*)
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

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=False*)
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

class fpsim.base.**BaseSim**(*pars*, ***kwargs*)
Bases: *fpsim.base.ParsObj*

The BaseSim class handles the dynamics of the simulation.

year2ind(*year*)

ind2year(*ind*)

ind2calendar(*ind*)

property npts
Count the number of points in timesteps between the starting year and the ending year.

property tvec
Create a time vector array at intervals of the timestep in years

property n

fpsim.calibration module

Define the Calibration class

class fpsim.calibration.**Calibration**(*pars*, *calib_pars=None*, *weights=None*, *verbose=True*, *keep_db=False*, ***kwargs*)

Bases: sciris.sc_utils.prettyobj

A class to handle calibration of FPsim objects. Uses the Optuna hyperparameter optimization library (optuna.org).

Note: running a calibration does not guarantee a good fit! You must ensure that you run for a sufficient number of iterations, have enough free parameters, and that the parameters have wide enough bounds. Please see the tutorial on calibration for more information.

Parameters

- **sim** (Sim) – the simulation to calibrate
- **calib_pars** (*dict*) – a dictionary of the parameters to calibrate of the format dict(key1=[best, low, high])
- **weights** (*dict*) – a custom dictionary of weights for each output
- **n_trials** (*int*) – the number of trials per worker
- **n_workers** (*int*) – the number of parallel workers (default: maximum)
- **total_trials** (*int*) – if n_trials is not supplied, calculate by dividing this number by n_workers
- **name** (*str*) – the name of the database (default: 'fpsim_calibration')
- **db_name** (*str*) – the name of the database file (default: 'fpsim_calibration.db')
- **keep_db** (*bool*) – whether to keep the database after calibration (default: false)
- **storage** (*str*) – the location of the database (default: sqlite)
- **label** (*str*) – a label for this calibration object
- **verbose** (*bool*) – whether to print details of the calibration
- **kwargs** (*dict*) – passed to cv.Calibration()

Returns A Calibration object

set_optuna_defaults()
Create a (mutable) dictionary with default global settings

configure_optuna(***kwargs*)
Update Optuna configuration, if required

validate_pars()
Ensure parameters are in the correct format. Two formats are permitted: either a dict of arrays or lists in order best-low-high, e.g.:

```
calib_pars = dict(
    exposure_factor           = [1.0, 0.5,  1.5],
    maternal_mortality_factor = [1,   0.75, 3.0],
)
```

Or the same thing, as a dict of dicts:

```
calib_pars = dict(
    exposure_factor           = dict(best=1.0, low=0.5,  high=1.5),
    maternal_mortality_factor = dict(best=1,   low=0.75, high=3.0),
)
```

run_exp(*pars*, *return_exp=False*, ***kwargs*)
Create and run an experiment

run_trial(*trial*)
Define the objective for Optuna

worker()
Run a single worker

run_workers()
Run multiple workers in parallel

remove_db()
Remove the database file if keep_db is false and the path exists.

make_study()
Make a study, deleting one if it already exists

calibrate(*calib_pars=None*, *weights=None*, *verbose=None*, ***kwargs*)
Actually perform calibration

summarize()

parse_study()
Parse the study into a data frame

to_json(*filename=None*)
Convert the data to JSON

plot_trend(*best_thresh=2*)
Plot the trend in best mismatch over time

plot_all()
Plot every point: warning, very slow!

plot_best(*best_thresh=2*)
Plot only the points with lowest mismatch

plot_stride(*npts=200*)
Plot a fixed number of points in order across the results

fpsim.defaults module

Define defaults for use throughout FPsim

fpsim.defaults.**pars**(*location=None*, ***kwargs*)
Function for getting default parameters.

Parameters

- **location** (*str*) – the location to use for the parameters; use 'test' for a simple test set of parameters
- **kwargs** (*dict*) – custom parameter values

Example:: pars = fp.pars(location='senegal')

fpsim.experiment module

Define classes and functions for the Experiment class (running sims and comparing them to data)

class fpsim.experiment.**Experiment**(*pars=None*, *flags=None*, *label=None*)
Bases: sciris.sc_utils.prettyobj

Class for running calibration to data

init_dhs_data()
Assign data points of interest in DHS dictionary for Senegal data. All data 2018 unless otherwise indicated Adjust data for a different year or country

extract_dhs_data()

pop_growth_rate(*years*, *population*)

initialize()

post_process_sim()

run_model(*pars=None*, *mother_ids=False*)
Create the sim and run the model

extract_model()

model_pop_size()

model_mcpr()

model_mmr()
Calculate maternal mortality in model over most recent 3 years

model_infant_mortality_rate()

model_crude_death_rate()

model_crude_birth_rate()

model_data_tfr()

extract_skyscrapers()

extract_birth_spacing()

extract_methods()

extract_age_pregnancy()

compute_fit(**args*, ***kwargs*)
Compute how good the fit is

post_process_results(*keep_people=False*, *compute_fit=True*, ***kwargs*)
Compare the model and the data

run(*pars=None*, *keep_people=False*, *compute_fit=True*, ***kwargs*)
Run the model and post-process the results

compare()
Create and print a comparison between model and data

summarize(*as_df=False*)
Convert results to a one-number-per-key summary format. Returns summary, also saves to self.summary.

Parameters **as_df** (*bool*) – if True, return a dataframe instead of a dict.

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

Parameters

- **filename** (*str*) – if None, return string; else, write to file
- **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
- **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 = calib.to_json()
calib.to_json('results.json')
```

plot(*axes_args=None*, *do_maximize=True*, *do_show=True*)
Plot the model against the data

class fpsim.experiment.**Fit**(*data*, *sim*, *weights=None*, *keys=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)
- mismatches: the sum of all the losses (a single scalar value per time series)
- mismatch: the sum of the mismatches – 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 (by default: 10 for deaths, 5 for diagnoses, 1 for everything else)
- **keys** (*list*) – the keys to use in the calculation
- **custom** (*dict*) – a custom dictionary of additional data to fit; format is e.g. {'my_output':{'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 cv.compute_gof() – see this function for more detail on goodness-of-fit calculation options

Example:

```
sim = cv.Sim()
sim.run()
fit = sim.compute_fit()
fit.plot()
```

compute()
Perform all required computations

reconcile_inputs(*verbose=False*)
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*, *do_show=True*)
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()
- **do_show** (*bool*) – whether to show the plot

fpsim.experiment.**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, skestimator='mean_squared_error') # Scikit-learn's MSE␣
↪method
e5 = compute_gof(x1, x2, as_scalar='median') # Normalized median absolute error --␣
↪highly robust
```

fpsim.experiment.**datapath**(*path*)

Return the path of the parent folder

fpsim.experiment.**diff_summaries**(*sim1*, *sim2*, *skip_key_diffs=False*, *output=False*, *die=False*)

Compute the difference of the summaries of two FPsim calibration objects, and print any values which differ.

Parameters

- **sim1** (*sim/dict*) – the calib.summary dictionary, representing a single sim
- **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:

```
c1 = fp.Calibration()
c2 = fp.Calibration()
c1.run()
c2.run()
fp.diff_summaries(c1.summarize(), c2.summarize())
```

fpsim.interventions module

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

class fpsim.interventions.**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

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 fpsim.interventions.**change_par**(*par*, *years=None*, *vals=None*, *verbose=False*)
Bases: *fpsim.interventions.Intervention*

Change a parameter at a specified point in time.

Parameters

- **par** (*str*) – the parameter to change
- **years** (*float/arr*) – the year(s) at which to apply the change
- **vals** (*any*) – a value or list of values to change to (if a list, must have the same length as years); or a dict of year:value entries

If any value is `'reset'`, reset to the original value.

Example:

```
ec0 = fp.change_par(par='exposure_factor', years=[2000, 2010], vals=[0.0, 2.0]) #␣
↪Reduce exposure factor
ec0 = fp.change_par(par='exposure_factor', vals={2000:0.0, 2010:2.0}) # Equivalent␣
↪way of writing
sim = fp.Sim(interventions=ec0).run()
```

`initialize`(*sim*)
Initialize intervention – this is used to make modifications to the intervention that can't be done until after the sim is created.

`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

`finalize`()
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)

class fpsim.interventions.**update_methods**(*year*, *eff=None*, *probs=None*, *matrix=None*, *verbose=False*)
Bases: *fpsim.interventions.Intervention*

Intervention to modify method efficacy and/or switching matrix.

Parameters

- **year** (*float*) – The year we want to change the method.
- **eff** (*dict*) – An optional key for changing efficacy; its value is a dictionary with the following schema:

  {method: efficacy} Where method is the method to be changed, and efficacy is the new efficacy (can include multiple keys).

- **probs** (*list*) – A list of dictionaries where each dictionary has the following keys:

  source (str): The source method to be changed. dest (str) The destination method to be changed. factor (float): The factor by which to multiply existing probability; OR value (float): The value to replace the switching probability value. keys (list): A list of strings representing age groups to affect. matrix (str): One of ['probs', 'probs1', 'probs1to6'] where:

  probs: Changes the specified uptake at the corresponding year regardless of state. probs1: Changes the specified uptake for all individuals in their first month postpartum. probs1to6: Changes the specified uptake for all individuals that are in the first 6 months postpartum.

apply(*sim*)
Applies the efficacy or contraceptive uptake changes if it is the specified year based on scenario specifications.

fpsim.scenarios module

Class to define and run scenarios

fpsim.scenarios.**make_scen**(**args*, ***kwargs*)
Alias for `fp.Scenario()`.

Store the specification for a single scenario (which may consist of multiple interventions).

This function is intended to be as flexible as possible; as a result, it may be somewhat confusing. There are five different ways to call it – method efficacy, method probability, method initiation/discontinuation, parameter, and custom intervention.

Args (shared): spec (dict): a pre-made specification of a scenario; see keyword explanations below (optional) args (list): additional specifications (optional) label (str): the sim label to use for this scenario year (float): the year at which to activate efficacy and probability scenarios matrix (str): which set of probabilities to modify for probability scenarios (e.g. annual or postpartum) ages (str/list): the age groups to modify the probabilities for

Args (efficacy): year (float): as above eff (dict): a dictionary of method names and new efficacy values

Args (probablity): year (float): as above matrix (str): as above ages (str): as above source (str): the method to switch from dest (str): the method to switch to factor (float): if supplied, multiply the [source, dest] probability by this amount value (float): if supplied, instead of factor, replace the [source, dest] probability by this value

Args (initiation/discontinuation): year (float): as above matrix (str): as above ages (str): as above method (str): the method for initiation/discontinuation init_factor (float): as with "factor" above, for initiation (None → method) discont_factor (float): as with "factor" above, for discontinuation (method → None) init_value

(float): as with "value" above, for initiation (None → method) discont_value (float): as with "value" above, for discontinuation (method → None)

Args (parameter): par (str): the parameter to modify years (float/list): the year(s) at which to apply the modifications vals (float/list): the value(s) of the parameter for each year

Args (custom): interventions (Intervention/list): any custom intervention(s) to be applied to the scenario

Congratulations on making it this far.

Examples:

```
# Basic efficacy scenario
s1 = fp.make_scen(eff={'Injectables':0.99}, year=2020)

# Double rate of injectables initiation
s2 = fp.make_scen(source='None', dest='Injectables', factor=2)

# Double rate of injectables initiation -- alternate approach
s3 = fp.make_scen(method='Injectables', init_factor=2)

# More complex example: change condoms to injectables transition probability for 18-
↪25 postpartum women
s4 = fp.make_scen(source='Condoms', dest='Injectables', value=0.5, ages='18-25',␣
↪matrix='pp1to6')

# Parameter scenario: halve exposure
s5 = fp.make_scen(par='exposure_factor', years=2010, vals=0.5)

# Custom scenario
def update_sim(sim): sim.updated = True
s6 = fp.make_scen(interventions=update_sim)

# Combining multiple scenarios: increase injectables initiation and reduce exposure␣
↪factor
s7 = fp.make_scen(
    dict(method='Injectables', init_factor=2),
    dict(par='exposure_factor', years=2010, vals=0.5)
)

# Scenarios can be combined
s8 = s1 + s2
```

class fpsim.scenarios.**Scenario**(*spec=None*, **args*, *label=None*, *year=None*, *matrix=None*, *ages=None*, *eff=None*, *probs=None*, *source=None*, *dest=None*, *factor=None*, *value=None*, *method=None*, *init_factor=None*, *discont_factor=None*, *init_value=None*, *discont_value=None*, *par=None*, *years=None*, *vals=None*, *interventions=None*)

Bases: sciris.sc_utils.prettyobj, sciris.sc_settings.dictobj

Store the specification for a single scenario (which may consist of multiple interventions).

This function is intended to be as flexible as possible; as a result, it may be somewhat confusing. There are five different ways to call it – method efficacy, method probability, method initiation/discontinuation, parameter, and custom intervention.

Args (shared): spec (dict): a pre-made specification of a scenario; see keyword explanations below (optional) args (list): additional specifications (optional) label (str): the sim label to use for this scenario year (float):

the year at which to activate efficacy and probability scenarios matrix (str): which set of probabilities to modify for probability scenarios (e.g. annual or postpartum) ages (str/list): the age groups to modify the probabilities for

Args (efficacy): year (float): as above eff (dict): a dictionary of method names and new efficacy values

Args (probablity): year (float): as above matrix (str): as above ages (str): as above source (str): the method to switch from dest (str): the method to switch to factor (float): if supplied, multiply the [source, dest] probability by this amount value (float): if supplied, instead of factor, replace the [source, dest] probability by this value

Args (initiation/discontinuation): year (float): as above matrix (str): as above ages (str): as above method (str): the method for initiation/discontinuation init_factor (float): as with "factor" above, for initiation (None → method) discont_factor (float): as with "factor" above, for discontinuation (method → None) init_value (float): as with "value" above, for initiation (None → method) discont_value (float): as with "value" above, for discontinuation (method → None)

Args (parameter): par (str): the parameter to modify years (float/list): the year(s) at which to apply the modifications vals (float/list): the value(s) of the parameter for each year

Args (custom): interventions (Intervention/list): any custom intervention(s) to be applied to the scenario

Congratulations on making it this far.

Examples:

```
# Basic efficacy scenario
s1 = fp.make_scen(eff={'Injectables':0.99}, year=2020)

# Double rate of injectables initiation
s2 = fp.make_scen(source='None', dest='Injectables', factor=2)

# Double rate of injectables initiation -- alternate approach
s3 = fp.make_scen(method='Injectables', init_factor=2)

# More complex example: change condoms to injectables transition probability for 18-
↪25 postpartum women
s4 = fp.make_scen(source='Condoms', dest='Injectables', value=0.5, ages='18-25',␣
↪matrix='pp1to6')

# Parameter scenario: halve exposure
s5 = fp.make_scen(par='exposure_factor', years=2010, vals=0.5)

# Custom scenario
def update_sim(sim): sim.updated = True
s6 = fp.make_scen(interventions=update_sim)

# Combining multiple scenarios: increase injectables initiation and reduce exposure␣
↪factor
s7 = fp.make_scen(
    dict(method='Injectables', init_factor=2),
    dict(par='exposure_factor', years=2010, vals=0.5)
)

# Scenarios can be combined
s8 = s1 + s2
```

update_label(*label=None*)
Ensure all specs have the correct label

class fpsim.scenarios.**Scenarios**(*pars=None*, *repeats=None*, *scens=None*, ***kwargs*)
Bases: sciris.sc_utils.prettyobj

Run different intervention scenarios.

A "scenario" can be thought of as a list of sims, all with the same parameters except for the random seed. Usually, scenarios differ from each other only in terms of the interventions run (to compare other differences between sims, it's preferable to use a MultiSim object).

Parameters

- **pars** (*dict*) – parameters to pass to the sim
- **repeats** (*int*) – how many repeats of each scenario to run (default: 1)
- **scens** (*list*) – the list of scenarios to run; see also `fp.make_scen()` and `Scenarios.add_scen()`
- **kwargs** (*dict*) – optional additional parameters to pass to the sim

Example:

```
scen1 = fp.make_scen(label='Baseline')
scen2 = fp.make_scen(year=2002, eff={'Injectables':0.99}) # Basic efficacy scenario
scens = fp.Scenarios(location='test', repeats=2, scens=[scen1, scen2])
scens.run()
```

add_scen(*scen=None*, *label=None*)
Add a scenario or scenarios to the Scenarios object

make_sims(*scenlabel*, ***kwargs*)
Create a list of sims that are all identical except for the random seed

make_scens()
Convert a scenario specification into a list of sims

run(**args*, ***kwargs*)
Actually run a list of sims

check_run()
Give a meaningful error message if the scenarios haven't been run

plot_sims(***kwargs*)
Plot each sim as a separate line across all senarios

plot_scens(***kwargs*)
Plot the scenarios with bands

plot_cpr(***kwargs*)
Plot the CPR with bands

analyze_sims(*start=None*, *end=None*)
Take a list of sims that have different labels and count the births in each

fpsim.sim module

Defines the Sim class, the core class of the FP model (FPsim).

class fpsim.sim.**People**(*pars*, *n=None*, ***kwargs*)
Bases: *fpsim.base.BasePeople*

Class for all the people in the simulation.

update_method()
Uses a switching matrix from DHS data to decide based on a person's original method their probability of changing to a new method and assigns them the new method. Currently allows switching on whole calendar years to enter function. Matrix serves as an initiation, discontinuation, continuation, and switching matrix. Transition probabilities are for 1 year and only for women who have not given birth within the last 6 months.

update_method_pp()
Utilizes data from birth to allow agent to initiate a method postpartum coming from birth by 3 months postpartum and then initiate, continue, or discontinue a method by 6 months postpartum. Next opportunity to switch methods will be on whole calendar years, whenever that falls.

update_methods()
If eligible (age 15-49 and not pregnant), choose new method or stay with current one

check_mortality()
Decide if person dies at a timestep

check_sexually_active()
Decide if agent is sexually active based either on month postpartum or age if not postpartum. Postpartum and general age-based data from DHS.

check_conception()
Decide if person (female) becomes pregnant at a timestep.

make_pregnant()
Update the selected agents to be pregnant

check_lam()
Check to see if postpartum agent meets criteria for LAM in this time step

update_breastfeeding()
Track breastfeeding, and update time of breastfeeding for individual pregnancy. Agents are randomly assigned a duration value based on a gumbel distribution drawn from the 2018 DHS variable for breastfeeding months. The mean (mu) and the std dev (beta) are both drawn from that distribution in the DHS data.

update_postpartum()
Track duration of extended postpartum period (0-24 months after birth). Only enter this function if agent is postpartum

update_pregnancy()
Advance pregnancy in time and check for miscarriage

reset_breastfeeding()
Stop breastfeeding, calculate total lifetime duration so far, and reset lactation episode to zero

check_maternal_mortality()
Check for probability of maternal mortality

check_infant_mortality()
Check for probability of infant mortality (death < 1 year of age)

check_delivery()
Decide if pregnant woman gives birth and explore maternal mortality and child mortality

update_age()
Advance age in the simulation

update_age_bin_totals()
Count how many total live women in each 5-year age bin 10-50, for tabulating ASFR

track_mcpr()
Track for purposes of calculating mCPR at the end of the timestep after all people are updated Not including LAM users in mCPR as this model counts all women passively using LAM but DHS data records only women who self-report LAM which is much lower. Follows the DHS definition of mCPR

track_cpr()
Track for purposes of calculating newer ways to conceptualize contraceptive prevalence at the end of the timestep after all people are updated Includes women using any method of contraception, including LAM Denominator of possible users includes all women aged 15-49

track_acpr()
Track for purposes of calculating newer ways to conceptualize contraceptive prevalence at the end of the timestep after all people are updated Denominator of possible users excludes pregnant women and those not sexually active in the last 4 weeks Used to compare new metrics of contraceptive prevalence and eventually unmet need to traditional mCPR definitions

init_step_results()

update()
Update the person's state for the given timestep. t is the time in the simulation in years (ie, 0-60), y is years of simulation (ie, 1960-2010)

class fpsim.sim.**Sim**(*pars=None*, *location=None*, *label=None*, *mother_ids=False*, ***kwargs*)
Bases: *fpsim.base.BaseSim*

The Sim class handles the running of the simulation

initialize(*force=False*)

init_results()

get_age_sex(*n*)
For an ex nihilo person, figure out if they are male and female, and how old

make_people(*n=1*, *age=None*, *sex=None*, *method=None*, *debut_age=None*)
Set up each person

init_people(*output=False*, ***kwargs*)
Create the people

update_methods()
Update all contraceptive method matrices to have probabilities that follow a trend closest to the year the sim is on based on mCPR in that year

update_mortality()
Update infant and maternal mortality for the sim's current year. Update general mortality trend as this uses a spline interpolation instead of an array

update_mothers()
Add link between newly added individuals and their mothers

apply_interventions()
Apply each intervention in the model

apply_analyzers()
Apply each analyzer in the model

run(*verbose=None*)
Run the simulation

store_postpartum()
Stores snapshot of who is currently pregnant, their parity, and various postpartum states in final step of model for use in calibration

to_df()
Export all sim results to a dataframe

plot(*do_save=None*, *do_show=True*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *fill_args=None*, *label=None*, *new_fig=True*)
Plot the results – can supply arguments for both the figure and the plots.

Parameters

- **dosave** (*bool*) – Whether or not to save the figure. If a string, save to that filename.
- **doshow** (*bool*) – Whether to show the plots at the end
- **figargs** (*dict*) – Passed to pl.figure()
- **plot_args** (*dict*) – Passed to pl.plot()
- **axis_args** (*dict*) – Passed to pl.subplots_adjust()
- **fill_args** (*dict*) – Passed to pl.fill_between())
- **label** (*str*) – Label to override default
- **new_fig** (*bool*) – whether to create a new figure (true unless part of a multisim)

plot_cpr(*do_save=None*, *do_show=True*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *fill_args=None*, *label=None*, *new_fig=True*)
Plot the results – can supply arguments for both the figure and the plots.

Parameters

- **dosave** (*bool*) – Whether or not to save the figure. If a string, save to that filename.
- **doshow** (*bool*) – Whether to show the plots at the end
- **figargs** (*dict*) – Passed to pl.figure()
- **plot_args** (*dict*) – Passed to pl.plot()
- **axis_args** (*dict*) – Passed to pl.subplots_adjust()
- **fill_args** (*dict*) – Passed to pl.fill_between())
- **label** (*str*) – Label to override default
- **new_fig** (*bool*) – whether to create a new figure (true unless part of a multisim)

plot_age_first_birth(*do_show=False*, *do_save=True*, *output_file='first_birth_age.png'*)

plot_people()
Use imshow() to show all individuals as rows, with time as columns, one pixel per timestep per person

log_daily_totals()

save_daily_totals()

class fpsim.sim.**MultiSim**(*sims=None*, *base_sim=None*, *label=None*, *n=None*, ***kwargs*)
Bases: sciris.sc_utils.prettyobj

The MultiSim class handles the running of multiple simulations

run(*compute_stats=True*, ***kwargs*)

compute_stats(*return_raw=False*, *quantiles=None*, *use_mean=False*, *bounds=None*)
Compute statistics across multiple sims

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 = fp.MultiSim.merge(msim1, msim2, base=True) mm2 = fp.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 that many chunks; if a list return chunks of that many sims

Returns A list of MultiSim objects

Examples:

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

msim = fp.MultiSim(fp.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=2) # Equivalent to inds=[[0,1,2], [3,4,5]]
```

remerge(*base=True*, ***kwargs*)
Split a sim, compute stats, and re-merge.

Parameters

- **base** (*bool*) – whether to use the base sim (otherwise, has no effect)
- **kwargs** (*dict*) – passed to msim.split()

Note: returns a new MultiSim object (if that concerns you).

to_df()
Export all individual sim results to a dataframe

plot(*do_show=True*, *plot_sims=True*, *fig_args=None*, *plot_args=None*, *plot_cpr=False*, ***kwargs*)
Plot the MultiSim

plot_cpr(**args*, ***kwargs*)
Plot the contraceptive prevalence rate

plot_method_mix(*n_sims=10*, *do_show=False*, *do_save=True*, *filepath='method_mix.png'*)
Plots the average method mix for n_sims runs

Parameters

- **n_sims** (*int*) – The number of sims you want to run to calculate average mix and standard deviation.
- **do_show** (*bool*) – Whether or not the user wants to show the output plot.
- **do_save** (*bool*) – Whether or not the user wants to save the plot to filepath.
- **filepath** (*str*) – The name of the path to output the plot.

plot_age_first_birth(*do_show=False*, *do_save=True*, *output_file='age_first_birth_multi.png'*)

fpsim.sim.**parallel**(**args*, ***kwargs*)
A shortcut to `fp.MultiSim()`, allowing the quick running of multiple simulations at once.

Parameters

- **args** (*list*) – The simulations to run
- **kwargs** (*dict*) – passed to multi_run()

Returns A run MultiSim object.

Examples:

```
s1 = fp.Sim(exposure_factor=0.5, label='Low')
s2 = fp.Sim(exposure_factor=2.0, label='High')
fp.parallel(s1, s2).plot()
msim = fp.parallel(s1, s2)
```

fpsim.utils module

File for storing utilities and probability calculators needed to run FP model

fpsim.utils.**set_seed**(*seed=None*)
Reset the random seed – complicated because of Numba

fpsim.utils.**bt**(*prob*)
A simple Bernoulli (binomial) trial

fpsim.utils.**bc**(*prob*, *repeats*)
A binomial count

fpsim.utils.**rbt**(*prob*, *repeats*)
A repeated Bernoulli (binomial) trial

fpsim.utils.**mt**(*probs*)
A multinomial trial

fpsim.utils.**fixaxis**(*useSI=True*, *set_lim=True*, *legend=True*)
Fix the plotting

fpsim.utils.**dict2obj**(*d*)
Convert a dictionary to an object – REFACTOR to use sc.dictobj()

fpsim.utils.**sample**(*dist='uniform'*, *par1=0*, *par2=1*, *size=1*, ***kwargs*)
Draw a sample from the distribution specified by the input. The available distributions are:

- 'uniform' : uniform distribution from low=par1 to high=par2; mean is equal to (par1+par2)/2
- 'normal' : normal distribution with mean=par1 and std=par2
- 'lognormal' : lognormal distribution with mean=par1 and std=par2 (parameters are for the lognormal distribution, *not* the underlying normal distribution)
- 'normal_pos' : right-sided normal distribution (i.e. only positive values), with mean=par1 and std=par2 *of the underlying normal distribution*
- 'normal_int' : normal distribution with mean=par1 and std=par2, returns only integer values
- 'lognormal_int' : lognormal distribution with mean=par1 and std=par2, returns only integer values
- 'poisson' : Poisson distribution with rate=par1 (par2 is not used); mean and variance are equal to par1
- 'neg_binomial' : negative binomial distribution with mean=par1 and k=par2; converges to Poisson with k=∞

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:

```
fp.sample() # returns Unif(0,1)
fp.sample(dist='normal', par1=3, par2=0.5) # returns Normal(=3, =0.5)
fp.sample(dist='lognormal_int', par1=5, par2=3) # returns a lognormally distributed␣
↪set of values with mean 5 and std 3
```

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 std of the lognormal distribution.

Negative binomial distributions are parameterized with reference to the mean and dispersion parameter k (see: https://en.wikipedia.org/wiki/Negative_binomial_distribution). The r parameter of the underlying distribution is then calculated from the desired mean and k. For a small mean (~1), a dispersion parameter of ∞ corresponds to the variance and standard deviation being equal to the mean (i.e., Poisson). For a large mean (e.g. >100), a dispersion parameter of 1 corresponds to the standard deviation being equal to the mean.

fpsim.version module

PYTHON MODULE INDEX

f

INDEX

F

G

I

K

L

M

N

P

R

S

T

U

V

W

Y